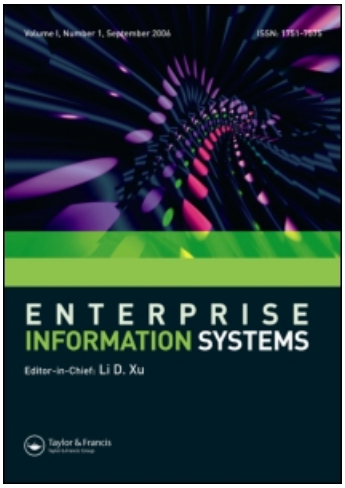


**Enterprise Information Systems**
Publication details, including instructions for authors and subscription information:
http://www.informaworld.com/smpp/title~content=t748254467

# A survey of software adaptation in mobile and ubiquitous computing

Konstantinos Kakousis[a]; Nearchos Paspallis[a]; George Angelos Papadopoulos[a]
[a] Department of Computer Science, University of Cyprus, Nicosia, Cyprus

Online publication date: 21 October 2010

**To cite this Article** Kakousis, Konstantinos , Paspallis, Nearchos and Papadopoulos, George Angelos(2010) 'A survey of software adaptation in mobile and ubiquitous computing', Enterprise Information Systems, 4: 4, 355 — 389
**To link to this Article: DOI:** 10.1080/17517575.2010.509814
**URL:** http://dx.doi.org/10.1080/17517575.2010.509814

*Enterprise Information Systems*
Vol. 4, No. 4, November 2010, 355–389

Taylor & Francis
Taylor & Francis Group

# A survey of software adaptation in mobile and ubiquitous computing

Konstantinos Kakousis*, Nearchos Paspallis and George Angelos Papadopoulos

*Department of Computer Science, University of Cyprus, Nicosia 1678, Cyprus*

(*Received 22 April 2009; final version received 16 July 2010*)

Driven by the vast proliferation of mobile devices and ubiquitous computing, dynamic software adaptation is becoming one of the most common terms in Software Engineering and Computer Science in general. After the evolution in autonomic and ubiquitous computing, we will soon expect devices to understand our changing needs and react to them as transparently as possible. Software adaptation is not a new term though; it has been extensively researched in several domains and in numerous forms. This has resulted in several interpretations of adaptation. This survey aims to provide a disambiguation of the term, as it is understood in ubiquitous computing, and a critical evaluation of existing software adaptation approaches. In particular, we focus on existing solutions that enable dynamic software modifications that happen on resource constrained devices, deployed in mobile and ubiquitous computing environments.

**Keywords:** dynamic software adaptation; context awareness; adaptation reasoning; adaptation acting; ubiquitous computing; mobile computing



## 1. Introduction

It has been almost two decades since Weiser (1993, 1999) first introduced the notion of ubiquitous computing. Although there are still ongoing arguments to which extent his vision for non-intrusive, seamless and constantly available human–computer interaction has been realised (Bell and Dourish 2007), this vision has undoubtedly guided significant research effort for the last 20 years. The envisioned unobtrusive interaction between smart objects and human users is nowadays possible due to the advances in hardware and software technologies in multiple research fields. Furthermore, several projects, both in the academia and the industry, have been concerned with – and are still investigating – the multi-disciplinary aspects of ubiquitous (or pervasive) computing.

The ubiquitous computing technology has already increased the demand for novel enterprise application and services, providing *anytime* and *anywhere* access. This applies particularly to users of mobile phones (Ljungstrand 2001) ('*the most prolific consumer product ever invented*' (Rubin 2008)). This is because existing enterprise services need to be adjusted to accommodate the limitations of mobile and ubiquitous computing (Hong *et al*. 2007) and also in order to take advantage of the features of new smartphone technology (Wright 2009). According to Strassner and Schoch (2003), the technical feasibility of such initiative is proved by five trends:

*Corresponding author. Email: kakousis@cs.ucy.ac.cy

ISSN 1751-7575 print/ISSN 1751-7583 online

DOI: 10.1080/17517575.2010.509814
http://www.informaworld.com

(1) The number of transistors on the same chip area doubles every 18 months (Moore's law (Moore 2000)). (2) Progress in communication technologies. Today mobile communication has become an essential part of our daily life. In the forthcoming century, it is expected that the society and economy will greatly depend on computer communications in digital format. Higher bandwidth, new generation mobile networks as well as new types of communication systems such as broadband wireless access systems, millimeter-wave LAN, intelligent transport systems (ITS) and high altitude stratospheric platform station (HAPS) are expected to further enhance the digital communication process (Ohmori *et al*. 2000). (3) Progress in sensor technology (sensors are getting smaller and cheaper so they can be easily embedded in everyday-use, hand-held devices). (4) New materials such as smart paper (E-Ink 2003) which can be used as user interaction media to enhance the communication in ubiquitous environments. (5) New concepts that model the infrastructure for everyday use of such smart items. At a larger scale, the examples of Singapore and Korea have already proved that the long expected technological advances (or trends) for enabling ubiquitous computing are already here (Bell and Dourish 2007). It is now up to enterprises and other involved parties (e.g. governments and organisations) how they will adjust their service and policies in order to evolve by adjusting to this new reality.

This article provides a survey and a thorough discussion of software adaptation techniques that enable developers of enterprise services to provide more flexible and capable ubiquitous computing applications. Software adaptation is a wide term, extensively studied in multiple disciplines and used to denote any kind of software modification at any phase throughout the whole lifecycle of a system. The interest on adaptable applications was increased after the proliferation of distributed and mobile computing and further accelerated with the emergence of ubiquitous and autonomic computing.

McKinley *et al*. (2005), claimed that foundations from autonomic computing along with advances in software engineering form the basis for most of the existing adaptation solutions. Furthermore, Satyanarayanan (2001) provided an excellent discussion on how pervasive computing builds on foundations of distributed computing, shares the same requirements with mobile computing and augments both by introducing new capabilities and requirements such as invisibility (i.e. minimal user interaction), smart spaces (i.e. integrating computing and building infrastructure), localised scalability (i.e. reduce interaction with context entities as the user moves away from them) and uneven conditioning (i.e. the transition period, where traditional computing applications and infrastructure are gradually replaced by others, introducing the features of pervasive computing).

Adaptation in ubiquitous computing is understood as the reactive process triggered by a specific event or a set of events in the context, with an ultimate goal to improve the QoS perceived by the end-user. Thus, the fundamental requirement for applications after the ubiquitous computing paradigm is the ability to *sense* their environment, reason upon context changes, and react (if necessary) accordingly. Inspired from the MAPE-K loop in the context of autonomic computing (Dobson *et al*. 2006), we define adaptation in ubiquitous computing as a closed loop (Figure 1) comprised of the following consecutive phases:

- *Context sensing and processing*: During this phase, all the data from the user context (such as the ambient noise, the current temperature and the user's

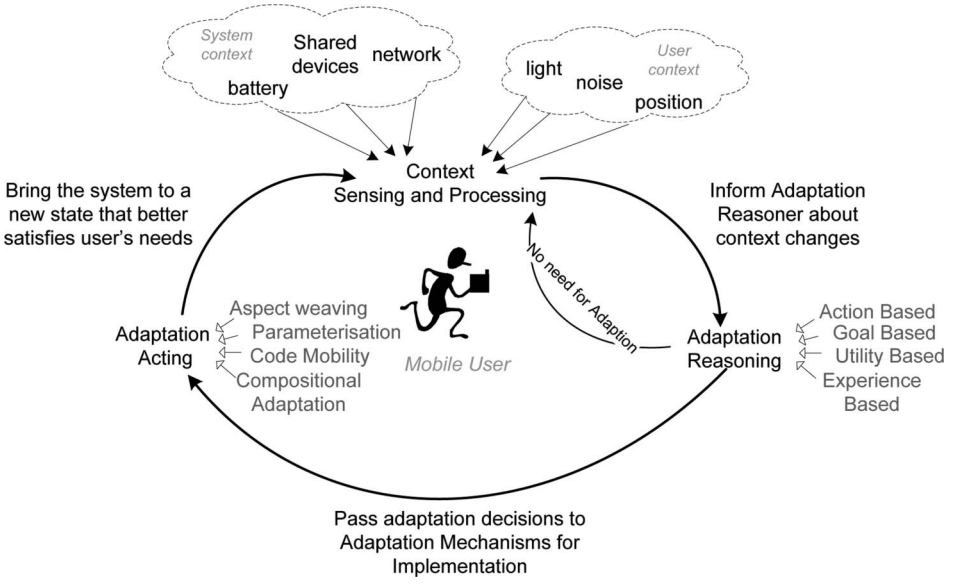


Figure 1. Adaptation loop in ubiquitous computing.

preferences) and the system context (such as the available computational resources and shared devices) are collected and often translated into high-level context events that might trigger a system adaptation. Although the focus of this survey is on *Adaptation reasoning* and *Adaptation acting* phases, we briefly discuss the *Context Sensing and Processing* phase in Section 2.

- *Adaptation reasoning and planning*: In this phase, the self-adaptive system is called to reason on the new context and decide what needs to be changed and how, in order to achieve the overall adaptation goal. Section 4 presents available adaptation reasoning techniques and evaluates their applicability to ubiquitous computing according to specific requirements.
- *Adaptation acting*: In this phase, appropriate adaptation mechanisms are used for implementing the adaptation decisions made by the reasoning process. Section 5 presents the most commonly used adaptation acting mechanisms and discusses their applicability to ubiquitous computing.

This survey reviews the state-of-the-art in software adaptation with emphasis on runtime modification of applications deployed on mobile devices. The rest of this article is organised as follows: Section 2 briefly describes the context sensing and processing phase of the adaptation loop. Section 3 gives background information on software adaptation and provides main classifications that restrict the scope of our discussion. Section 4 evaluates adaptation reasoning approaches with regard to their applicability to ubiquitous computing. Section 5 presents available mechanisms for implementing adaptation decisions and discusses their suitability to mobile, context-aware systems. Section 6 discusses some research challenges and issues in dynamic software adaptation and identifies possible future directions. Finally, Section 7 concludes this article.

## 2. Context sensing and processing

Context-aware computing has been actively studied since at least the mid-1990s (Schilit *et al*. 1994), shortly after Weiser first introduced the concept of ubiquitous computing (Weiser 1993, 1999). Initially, the research was concentrated on stand-alone context-aware applications, especially location-aware ones such as tour guides and active badges (Want *et al*. 1992, Long *et al*. 1996, Pascoe 1997). However, as the interest for context-awareness was extended to cover a much wider domain of context types and uses, the research was naturally shifted towards building frameworks providing support for general context-awareness. Perhaps the most popular of those is the Context Toolkit (Dey 2001) while, although more recent, approaches like CML (Henricksen and Indulska 2005, Henricksen *et al*. 2006) and COSMOS (Conan *et al*. 2007, Rouvoy *et al*. 2008b) are also widely referenced.

While many definitions of context are available in the literature, the one by Dey is the most widely cited one: '*Context is any information that can be used to characterise the situation of an entity; an entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and application themselves*' (Dey 2001). After that, Dey has also defined context-aware behaviour as follows: '*A system is context-aware if it uses context to provide relevant information and/or services to the user, where relevancy depends on the user's task*'. Although not explicitly indicated by these definitions, software adaptation is one of the most common methods for realising context-aware systems, as it allows varying the system behaviour in runtime. More specifically, compositional adaptation is particularly important for mobile and pervasive systems, as it allows for better resource utilisation (Paspallis *et al*. 2008b).

In software systems, which is the main focus of this article, context-aware behaviour is realised in different ways. While in most cases context-awareness implies *context sensing, reasoning* and *reaction*, in some approaches these aspects are clearly separated and different components are used for context sensing and processing, and others for actuation. For example, in the Context Toolkit, the proposed abstractions include: *widgets, aggregators, interpreters, services* and *discoverers*. Widgets are components that are responsible for acquiring context information directly from sensors. Aggregators can be thought of as meta-widgets, taking on all capabilities of widgets. They also provide the ability to aggregate context information of real-world entities such as users or places and act as a gateway between applications and widgets. Interpreters transform low-level information into higher-level information that is more useful to applications. Services are used by context-aware applications to invoke actions using actuators. Finally, discoverers are used to locate suitable widgets, interpreters, aggregators and services (Dey 2001). On the other hand, in the MUSIC middleware (IST-MUSIC 2009), the context middleware leverages context plug-ins for collecting, processing and elevating context information. The actual exploitation and use of the context information is left to the application developers (Paspallis *et al*. 2009).

Although context-awareness techniques are tightly related to software adaptation, this article focuses primarily on adaptation reasoning, planning and actuating. Extensive reviews of the motivation, challenges and state-of-the-art results focusing on context-awareness are available in the literature (Chen and Kotz 2000, Strang and Linhoff-Popien 2004, Baldauf *et al*. 2007, Bolchini *et al*. 2009, Paspallis 2009).

## 3. Adaptation background

The Oxford Dictionary of Sciences defines adaptation as '*any change in the structure or functioning of an organism that makes it better suited to its environment*'. The parallelism in computer science should be straight forward: it is any kind of structural, functional or behavioural modification of a software component, with the aim of better fitting to a changing environment and satisfying a high-level overall objective. More formally, yet generally enough, Capra *et al.* (2001) defined adaptation as '*the ability of the application to alter and reconfigure itself as a result of (i.e. in reaction to) context changes [...] to deliver the same service in different ways when requested in different contexts and at different points in time*'. The need for adaptation in mobile and ubiquitous computing becomes a fundamental requirement, since frequent context changes may affect the functional as well as the QoS properties of the system. This section aims to precisely define the scope of our discussion by presenting common classifications in software adaptation.

A first, coarse-grained categorisation is between *functional* and *extra-functional* adaptation. The former refers to modifying or correcting the system's functionality, while the latter targets changes in the system's extra-functional properties, such as its performance, accuracy, reliability and other QoS aspects. On the same axis, Ketfi *et al.* (2002) proposed a finer grained categorisation based on the overall goal of the adaptation. *Corrective adaptation* occurs when a malfunction is detected to a subcomponent of an application, which is replaced by a new version with exactly the same functionality but without the faulty part. *Adaptive adaptation* reactively adapts the system in response to some context changes that might affect its behaviour. *Extending adaptation* deals with the need for adopting new functionalities or services that have not been considered during development or deployment time. Finally, *Perfective adaptation* aims to improve an application even if it runs correctly. If, for example, the performance, security or any other extra functional property of the system is not considered optimal the perfective adaptation may be triggered. The overall objective of a self-adaptive application must be clear because it affects the selection of suitable adaptation reasoning approaches and implementation mechanisms.

Secondly, we distinguish between *static* and *dynamic* software adaptation. The former refers to any action that takes place pre-runtime, while the latter refers to any reconfiguration that occurs at runtime. More specifically, Canal *et al.* (2004) and Becker *et al.* (2006), classify software adaptation according to the moment in which the adaptation need is detected. The identified categories are as follows:

- *Requirements adaptation*: Occurs pre-runtime, usually when the requirements of a system are extended to meet new properties, or when a system specification must be adapted to meet the requirements of the new system. Requirements adaptation may trigger both functional and non-functional changes.
- *Design time adaptation*: Also occurs pre-runtime and whenever an analysis of the system architecture indicates a mismatch between two constituent components. For example, component mismatches may occur due to the use of different platforms, operating systems or frameworks, due to different names of methods, services and parameter types, due to differences between the provided and the expected quality of service (QoS) or during the interaction between connected components. Canal *et al.* (2008) and Becker *et al.* (2006)

provide a detailed description of the various mismatch types that may occur in component based software engineering.

- *Dynamic adaptation*: Occurs after the adaptable system has been deployed and refers to the situation where some running pieces of software need to be adapted in order to better fit to their execution environment. In contrast to static adaptation, in dynamic adaptation the components to be adapted, as well as the adaptation steps, are unknown until the moment of the adaptation. Dynamic adaptation is closely related to context-aware systems and is often limited to non-functional adaptation.

In ubiquitous computing, the provision of the expected functionality with the desired quality requires composite systems that can be adapted dynamically. (i.e. capable of reasoning about context changes and reconfiguring their behaviour in a way that satisfies a well-defined objective). Therefore, in this survey we restrict our discussion on *Dynamic* software adaptation with an overall objective of maximising user satisfaction, or in other words, minimising mismatches in the QoS. Static adaptation on the other hand, has different objectives, such as maximising code reuse and facilitating software maintenance. Static adaptation has been extensively examined elsewhere (Canal *et al*. 2004,Becker *et al*. 2006, Canal *et al*. 2008, Kell 2008) and it is outside the scope of this survey.

Dynamic adaptable systems can be further distinguished to *reactive* and *proactive*. The former refers to the ability of a system to alter and reconfigure itself in response to a context change. Reactively adaptable applications are expected to define the context elements that are interested in, identify relevant context changes and specify suitable adaptation and reconfiguration strategies to be applied if such changes occur. Conversely, proactive adaptation, driven by pervasive computing, refers to environments acting proactively by suggesting adaptation possibilities or directly adapting applications. Systems supporting proactive adaptation mainly target minimisation of user's distraction through environments composed of pervasive blocks that seamlessly interact and adapt to the user's needs. Several projects support reactive (David and Ledoux 2003), proactive (Garlan *et al*. 2002, Popovici *et al*. 2003) or hybrid (Capra *et al*. 2005, 2003) adaptation functionality. Generally speaking, proactive approaches require a greater synergy between hardware infrastructure (i.e. monitoring sensors), middleware components and user applications but they fit much better to the objectives of pervasive computing for an environment that satisfies user's context needs. Modern systems are expected to support both reactive and proactive adaptation to avoid limiting their adaptation capabilities.

Finally, from a technical perspective, adaptation methodologies are generally identified either as *parameter-based* or *compositional*. The former refers to techniques that affect the system's behaviour through property-tuning or variable modification without changing any structural or algorithmic part of the system. On the contrary, any approach that targets the system's behaviour through structural, geographical, interface or implementation modifications can be labelled as compositional adaptation.

## 4. Adaptation reasoning

We define adaptation reasoning as the adaptation process triggered by a change (or a set of changes) in the context and which aims to decide when and how often

adaptation is needed, which alternative best satisfies the adaptation overall objective and what adaptation operations are necessary in order to bring the system into the next configuration state in a safe and timely manner. In some cases any application variant satisfying specific criteria may be acceptable, while in some other cases only the optimal configuration (i.e. the one that maximises the overall objective) is required. Adaptation reasoning techniques applied in ubiquitous computing are mainly adopted from correlated fields, such as *autonomic computing, artificial intelligence* (AI) and *machine learning*.

We mainly classify adaptation reasoning techniques depending on whether adaptation (re)configurations are predefined (i.e. global configurations are pre-constructed at design time) or generated and evaluated at runtime. Predefined configuration is much easier since the adaptation reasoner only has to choose from a fixed set of global configurations. However, if new components become available or new applications are installed, the set of global configurations must be updated, something that might cause unacceptable delays. In addition, depending on the number of variation points in a system (i.e. the possible alternative system realisations) constructing every possible adaptation configuration at design time may turn into a particularly heavy task for application developers. Representative approaches of this category are *action-* and *goal-based* adaptation.

Dynamic reasoning on the other hand, is more flexible. It allows runtime modification of the available system realisations without further re-evaluation of adaptation configurations since they are all constructed and evaluated dynamically. Reasoning techniques, based on utility functions and AI, belong to the above category. Dynamic reasoning is considered more appropriate for systems with multiple variation points and frequent context changes but also requires more resources at runtime. Thus, choosing a reasoning approach depends on the adaptation objective, the available resources, the adaptation frequency and the number of variation points.

For the rest of this section, we review adaptation reasoning techniques and we critically evaluate their applicability to ubiquitous computing according to the following requirements:

(1) *Resource efficiency*: Resources of mobile devices are limited and precious, this criterion evaluates the applicability of a reasoning approach according to its demands for resources.
(2) *Coverage of the context value domain*: This requirements refers to the ability of a reasoning approach to operate and produce decisions on any subset of the context value domain.
(3) *Ease of building*: This criterion evaluates the applicability of an adaptation reasoning approach with regard to the development effort needed for implementing self-adaptive applications using this approach.
(4) *Adaptation openness*: Open self-adaptive software is the one that can be extended with new functionality, unknown at design time (e.g. newly discovered services or adaptable entities). Consequently, new adaptation behaviour and alternatives can be introduced. This criterion evaluates the ability of an adaptation reasoning approach to decide based on new adaptation alternatives as well.
(5) *Adaptation coordination*: One of the biggest challenges for an adaptation reasoning approach is its ability to deal with more than one self-adaptive

application that run concurrently on the same device. This criterion evaluates its ability to coordinate adaptation and resolve conflicts that may occur due to contradicting adaptation goals.

(6) *Evolvability*: This criterion refers to the ability of an adaptation reasoning approach to dynamically improve itself based on previous experience or feedback received from the environment.

(7) *Traceability*: This criterion refers to the ease of tracing, and producing human-readable explanations on the logic behind an adaptation decision. Traceability increases user trust and boosts technology adoption.

The above requirements are not presented in any particular order but their numbering is used for reference in the rest of this section. The requirements have been derived after an extensive review of existing approaches in adaptation reasoning for ubiquitous computing. These requirements are specific to the adaptation reasoning approach and represent only a small subset of the requirements that a complete context-aware, self-adaptive system should satisfy for the purposes of ubiquitous computing. In practice, many solutions depend on more complex platforms, typically involving a middleware layer, which usually supports several requirements, such as platform independence, fault-tolerance and enhanced security. For instance, with regard to the platform independence requirement, existing solutions are explicitly designed so that they hide the platform-specific properties of the underlying (heterogeneous) devices. As an example, we refer to the MUSIC (IST-MUSIC 2009) middleware which does exactly this while implementing a utility-based approach. Nevertheless, the overall architecture could be easily adapted to using any of the discussed adaptation reasoning approaches.



### *4.1. Action-based adaptation*

Action-based (or rule-based) adaptation is undoubtedly the most popular approach for defining the behaviour of self-managing systems in several domains related to networks and distributed systems. It is based on the notions of *states* and *actions* in the following way: at any point in time $t$, a system can be characterised as being in a state $S$ where if a policy $p$ is satisfied, an action $a$ will cause a probabilistic or deterministic transition of the system to a new state $S'$. In other words, a set of IF(Condition) – Then(Action) rules define precisely how the system will adapt in certain situations (Kephart and Das 2007).

Besides several works that use action policies for networks and systems optimisation (Lutfiyya *et al*. 2001, Lymberopoulos *et al*. 2002), Efstratiou *et al*. (2002) proposed an adaptation platform for mobile systems driven by variations of action policies based on *event calculus* (Shanahan 1999). Event calculus-based policies, formulated in terms of conditions and actions define exactly the adaptive behaviour of the corresponding application. Conditions are logical expressions that can be evaluated either to true or false and actions specify a set of adaptation methods that should be invoked if the condition evaluates to true.

Furthermore, the concept of event-action rules has been also deployed for expressing dynamic reconfigurations over component-based architectures. An early example is the DART (Raverdy and Lea 1999) project where an adaptation manager applies adaptation policies that are triggered by specific events, and generated based on system statistics and user requirements. Each policy is registered for one or more

adaptation events and whenever an event is received, the manager finds the policies to be called and executes them. Finally, to resolve conflicts among policies (several policies may register for the same event), suitable priorities are assigned to each policy.

In the same context, Chisel (Keeney and Cahill 2003) and SAFRAN (David and Ledoux 2003) proposed self-adaptive components driven by action policies that are handled separately from the functional parts of an application. In Chisel, there are policy specification documents containing text-based declarative representations of policy rules which in response to relevant context changes may trigger an adaptation. Following the same (Event–Condition-Action) pattern SAFRAN provides a context-aware component (WildCAT) that detects incoming events which correspond to reconfiguration actions written in a simple scripting language (FScript). An adaptation manager allows dynamic attachment of SAFRAN policies to individual components and is responsible for their execution.

An obvious limitation of rule-based approaches is the imposed binary decision logic since each rule is either evaluated to true or false. This limitation becomes more obvious in dynamic environments and may lead to *low coverage of the context value domain* (2). Fuzzy logic (Wang 1996) and fuzzy rules introduce a more human-like way of thinking. They allow for multi-valued logic with different degrees of truth. Thus, instead of having IF-THEN-ELSE rules we simply have IF-THEN rules (that can be simultaneously valid), or equivalent constructs, such as fuzzy associative matrices that allow multiple rules to be evaluated and assigned different degrees of truth for the same case.

Typically, rule-based approaches appear to be powerful in pervasive computing since they are *not resource demanding* (1) and they precisely define the adaptation behaviour of self-adaptive systems (Lutfiyya *et al*. 2001), even in the context of multiple applications running concurrently (5) (Efstratiou *et al*. 2002). Furthermore, they provide *high traceability* (7) of adaptation decisions, since they favour easier production of human-readable explanations of the reasoning behind a system reaction.

On the other hand, application developers need to explicitly describe, at design time, the adaptive reaction of the system in response to any relevant context fluctuation that may trigger an adaptation at runtime. In other words, every possible context change that may trigger an adaptation must be reflected by at-least one adaptation rule. Therefore, the more context elements we have, the more conditions and rules need to be specified. In the ever-changing, multi-dimensional context space of ubiquitous computing, identifying every possible adaptation action, in advance, is not an easy task and implies *high development effort* (3). It requires policy makers that are intimately familiar with low-level details of the system and specialised policy languages capable of expressing dependencies between adaptation and context. In most cases, policy files are written against the specifications of particular systems and do not provide the flexibility required for the heterogeneous environment of ubiquitous computing. Finally, *dynamic modification* and *evolution* of rules increases complexity and often requires recompilation (4), (6).

### *4.2. Goal-based adaptation*

Instead of exactly specifying the system's transition from the current state to the next one, in the form of IF-THEN actions, goal-based policies specify either a single

desired state or the criteria that characterise a whole set of desired states. Thus, it is the system's responsibility to decide which adaptation actions will cause a transition to the desired state. This relieves human administrators from the hard task of specifying adaptation actions, which requires low-level knowledge of the system's functionality (3). Goal-based approaches provide the flexibility to define the desired behaviour using high-level goals that are closer to the human way of reasoning. This enables goal-based approaches to provide *good traceability support* (7). This sort of flexibility does not come for free though: introducing rational behaviour into the system requires sophisticated planning and modelling algorithms. In particular, most policy-driven management systems require a mathematical model of the system, modules that monitor specific system parameters and prediction modules that provide estimates for the system's workload and resource needs.

Goal-based approaches are *efficient in terms of resource needs* (1) and have been used by several control-theoretic approaches for managing computing systems (Kephart and Walsh 2004). However, we argue that they have major drawbacks. For instance, goal-based policies fail to catch dependencies between adaptation and goals while in the case of limited resources, goal policies are prone to conflicts if the system cannot satisfy all the goals simultaneously. Goal-based approaches do not provide resolution mechanisms between contradictory goals, nor do they support state comparison mechanisms in the case that more than one state can fulfil the same goal (2), (5). Finally, adding new goals or removing existing ones usually leads to significant changes on the decision mechanism and might require recompilation (4), (6). After all, goal policies have been superseded by utility functions which, as Kephart and Das (2007) claim, are natural extensions of goal policies, but with several advantages.

### *4.3. Utility functions*

The notion of utility function was originally introduced in the fields of economics and AI. Recently, autonomic and ubiquitous computing have also adopted utility-based approaches for enabling adaptation reasoning. Generally speaking, utility functions are tools for measuring preferences. In particular, they are mathematical artifacts that map each possible system state, or alternative implementations, to a real scalar value. According to this value, a rational agent (a human or a software entity) can select the alternative implementation or system state that maximises the utility. Utility functions are considered an improved extension of goal policies, because instead of classifying system states to desired and undesired they assign scalar values (i.e. scores) to them indicating their applicability to the current context. Utility functions allow elimination of policy conflicts when multiple goals can be met simultaneously. In addition, while goal-oriented approaches only indicate feasible solutions, utility functions provide runtime determination of the optimal adaptation alternative. Finally, they provide a finer degree of expression that includes multiple adaptation aspects incorporating the user preferences as well.

Utility functions have been applied in several works, especially QoS-based, for measuring the suitability of adaptation alternatives in fluctuating environments. An early example in the area of mobile computing was the Odyssey (Noble and Satyanarayanan 1999), and subsequently, the Aura (Garlan *et al.* 2002) projects. These projects deal with mobile data access and target resource utilisation of mobile devices through utility-based adaptation. Each data item has a current *reference copy*

which is the most complete and detailed version of the data item under the current resource availability. Ideally, the reference copy is always presented to the mobile user. However, when resources become scarce, the item will be degraded in some way. A quality metric, very similar to utility – called *fidelity* – measures the degree of similarity between the original item and the reference copy. The operating system is responsible for monitoring resource availability and notifying affected applications of relevant changes in those resources. Upon notification, applications enter an adaptation decision loop which results in a new fidelity for the current resource situation and then enters a state where it waits for the next notification.

Capra *et al.* (2002) used a microeconomic-inspired mechanism for conflict resolution in a mobile setting. This mechanism relies on a particular type of sealed-bid auctions for conflict resolution while utility functions are used for incorporating user preferences in the resolution mechanism.

In MADAM (Geihs *et al.* 2009a) and MUSIC (IST-MUSIC 2009), applications are assembled through a recursive component composition process and variability is achieved by plugging into the same component type different component implementations with the same functional behaviour. After the MDA development methodology, adaptation is captured in platform-independent adaptation models which are transformed into a representation that the middleware can access and use at runtime. Finally, the middleware uses utility functions to calculate utility scores for each application variant. The highest utility score indicates the most suitable variant for the current context and it is selected for adaptation.

Similarly, Gjørven *et al.* (2006) proposed a middleware managed adaptation approach based on utility functions and service reflection. Adaptation is based on service planning which is responsible for searching and evaluating among alternative service mirrors and in the end selecting the one with the highest utility score. Utility rankings are calculated by utility functions which take as input QoS predictions that encode developers' knowledge about the QoS implementation.

Finally, Paspallis *et al.* (2008a) propose a multi-dimensional, utility-based model which attempts to simulate the user's adaptation reasoning mechanism. Utility functions are formulated over a set of pre-defined adaptation aspects, considered important for the adaptation reasoning process. The utility scores are calculated independently for each dimension, and the overall utility is computed as their weighted sum. Weight values are used to reflect the importance of each adaptation aspect for the targeted user.

Utility functions have several advantages over the rest of the approaches discussed in this article. According to Geihs *et al.* (2009a), Kephart and Das (2007), they appear to fit very well in the context of ubiquitous computing. Besides their ability to produce scalar decisions based on any subset of the context value domain (2), they can provide an effective decision mechanism for systems with multiple self-adaptive applications running concurrently. The applications may run on the same or different devices. Utility functions can resolve conflicts between contradicting adaptation goals since they can be defined as parts of a more generic utility function that aggregates the adaptation goals of several parts of the same system. Khan *et al.* (2009) proposed a solution for such a modular adaptation decision-making mechanism based on utility functions (5). In addition, utility functions support very *high-adaptation openness* (4) since they can incorporate in their decision mechanism newly discovered adaptable entities (unknown at design time), given that they provide the required QoS properties. Finally, as Kakousis *et al.* (2008)

suggested, weighted utility functions in combination with optimisation techniques can support *high evolvability* (6) of the mechanism, based on received user feedback.

On the other hand, defining utility functions, especially for complex applications, might be a difficult task for developers (3). Utility functions are based on variant properties prediction, current context situation and user preferences. Thus, application developers need to specify application variants in great detail and formulate every possible adaptation aspect in mathematical equations. Because of the complex adaptation reasoning, utility functions provide *very poor traceability* feedback as well (7). Furthermore, utility functions assume calculation and evaluation of each variant's utility. Therefore, utility functions present an *average resource efficiency* (1) and additional mechanisms (such as architectural constraints (Khan *et al*. 2008)), as well as, complicated heuristics are needed for discarding invalid variants, in order to avoid scalability issues (Khan *et al*. 2009) (see Section 6.2).



### *4.4. Adaptation reasoning by experience*

The adaptation reasoning techniques examined so far are either based on predefined rules and actions (action-based and goal-based adaptation) or on utility functions. Such approaches are not always capable of specifying adaptation rules that cover the whole context space and even when they do, developers cannot be sure that every adaptation action programmed in design time will be the optimal for the current context situation at runtime. Even if the context state is evaluated at runtime, the adaptation reasoning remains the same, no matter if the adaptation decision does not maximise the user's utility. In other words, we indicate the lack of *remembering* and *learning* from previous experiences. Gathering knowledge from system's previous decisions (successful or not) can be valuable for improving the adaptation behaviour of the system. *Case-based reasoning* (CBR) and *machine learning* approaches are capable of learning from their previous decisions and improve their behaviour based on collected feedback. Even though such approaches suffer from some obvious limitations (they usually need long training periods, extensive numbers of samples and occasionally make 'bad' decisions in order to explore the quality of alternative actions), in certain cases their contribution to dynamic adaptation may lead to improved adaptation reasoning.

#### *4.4.1. Case-based reasoning*

Instead of solving problems from scratch, by chaining generalised rules CBR follows an approach much closer to the human-problem solving reasoning. This gives a *high traceability* (7) advantage to CBR over other techniques. It is based on the assumption that similar problems have similar solutions and certain problems tend to recur. Solutions in CBR are generated based on previously stored, relevant cases and knowledge. This means that they need long training periods and extensive numbers of samples before they can produce meaningful adaptation decisions. However, knowledge can be also retrieved from external case bases, a process known as *multicase-based reasoning* (Leake 1996).

CBR is often classified to *interpretive* and *problem-solving*. The former uses prior knowledge to classify or characterise new situations while problem-solving attempts to apply prior solutions to new problems (Leake and Sooriamurthi 2002). A typical CBR solver, after receiving a user's query, it assesses and retrieves from the case base

the most similar problem descriptions along with their corresponding solutions. The final step is to adapt correlated solutions to solve the new problem. Adapting available solutions in CBR is considered to be one of the most difficult parts and several approaches have been proposed for enabling automatic case adaptation. Whether effective adaptation of relevant cases can be achieved it is still a debate (Manzoni *et al*. 2007), nevertheless the development process involves complex knowledge engineering tasks and the effort is significantly increased (3).

Case adaptation types in CBR can be classified in three main categories (Passone *et al*. 2006): *Null, transformation* and *generative* adaptation. Null adaptation is based on the *Nearest Neighbour* (NN) technique and simply applies the most relevant solution from the case base without any reconfigurations. Transformation adaptation performs structural transformations of previous solutions based on rules specified by domain experts or learnt using an induction algorithm and, finally, Generative adaptation generates solutions of the problem from scratch. The simplest adaptation strategy consists of using general or domain specific adaptation rules. However, it requires deep knowledge and manual maintenance of rules through updates or context changes. Multiple alternatives have been proposed but in most cases they are too complicated or too specific.

Although CBR may produce interesting results when based on a significant amount of previously stored relevant cases, in ubiquitous computing reasoning it may lead to non-acceptable adaptation decisions when it has insufficient sample input. In particular, the assumption of CBR that similar problems have similar solutions and certain problems tend to recur, barely fits to the concept of ubiquitous computing where almost identical context changes may require completely different adaptation actions. Therefore, *low coverage of the context value domain* (2) is an important disadvantage of CBR while reasoning on long stored data also needs extra resources (1). Additionally, when more than one application takes part to the adaptation process or new adaptation alternatives are discovered, it becomes more difficult to find relevant cases that will lead to accurate decisions (4), (5). On the other hand, CBR is considered an effective AI reasoning technique, suitable for routine or novel problems. It is *highly evolvable* (6) since it can improve itself, by learning from successful decisions of the past, and by avoiding prior failures.

### *4.4.2. Reinforcement learning*

Reinforcement learning (RL) is an unsupervised learning mechanism that selects specific actions in uncertain environments that maximise an overall, long-term goal. Main characteristics of RL are trial-and-error search and long-term rewards, meaning that RL systems learn from their experiences and prefer to choose actions that maximise the overall goal over a time horizon, even if that means poor instant payoff. Optimal reinforcement decisions are based on previous actions that are both, rewarding and representative of the whole context space. However, before a representative set of rewarding actions is formulated, new actions – that possibly have bad rewards – may need to be explored. In RL this is known as the exploitation versus exploration dilemma and refers to how we can exploit actions proved to be rewarding in the past, but in the same time explore new cases (both with good and bad rewards) in order to extend the sample data and thus improve future decision making (Sutton and Barto 1998). That said, RL is characterised by its *high evolvability* (6) and self-improvement while adaptation openness can be achieved

after some reinforcements (4). On the other hand, the *coverage of the context value domain* might not be the expected (2), if not sufficient trial-and-error attempts have been processed. Furthermore, extra processing and storage resources, already scarce in mobile devices, are needed for the RL process (1).

Typical reinforcement learning problems are modelled as *Markov Decision Processes* (MDPs) and consist of a set of environment states $S$, a set of actions $A$ and a set of scalar real values, used as rewards. At each point in time $t$ a system is characterised as being in state $S_t \epsilon S$ from where it can take $A(S_t)$ possible actions. If $a \epsilon A(S_t)$ is chosen then the system is transferred in state $S_{t+1}$ and a reward $r_{t+1}$ is generated. Based on these interactions a policy $\pi: S \rightarrow A$ is formulated in a way that maximises the overall reward that can be accumulated over the future, starting from the current state. Optionally, RL strategies are based on internal models of the environment and calculate the optimal policy based on these models. Models are used for planning future actions of a system before they are actually experienced and can be beneficial in cases where acquiring real-world experience is expensive. Model-free approaches, on the other hand, learn directly from experience, are faster and require less storage.

Dowling *et al.* (2005), proposed a technique for solving optimisation problems using distributed reinforcement learning agents. The technique is called *Collaborative Reinforcement Learning* (CRL) and extends traditional RL with various feedback models, including negative feedbacks that demonstrate an agent's view of its neighbourhood and a collaborative feedback model that allows agents to exchange gained knowledge between each other. In CRL system optimisation, problems are decomposed to *Discrete Optimisation Problems* (DOP) and distributed to be solved by collaborating RL agents. Thus, in CRL the set of possible actions that an agent can execute is enriched with DOP actions which try to solve the problem locally, delegation actions that delegate the solution of a DOP to a neighbour and discovery actions that can be executed by an agent in any state in order to find new neighbours. Delegation can occur either because a problem cannot be solved locally or the estimated cost of solving it remotely is much less. Although we are not aware of any research attempt towards this direction, CRL can be used for better coordination of systems with more than one self-adaptive application (5).

Finally, Charvillat and Grigoraş (2007), have used RL in dynamic multimedia adaptation of systems that have to deal with frequent context changes, limited resources and uncertainty in interaction with the user. An adaptation agent perceives the characteristics of the environment which integrates the user, his mobile terminal, the network and various contents (such as documents, media, etc) and based on prior knowledge and training data produces the adaptation decision (Figure 2).

The agent may manage, transform or prioritise contents, make bandwidth or resources allocation, act on behalf of the user or simply do nothing. The agent generates and gradually improves a decision policy capable of choosing the best alternative action from each state.

Overall, RL and its variations can fit very well to distributed adaptation of component based systems, acting in resource constrained environments. The power of RL can be exploited in order to allow such systems to self-organise their component deployment configuration in a way that best utilises the resource availability in local and distributed devices. In particular, through an MDP we can model alternative component delegation actions from each possible state. Consequently, we can achieve optimal decisions on (re)deployment of components in

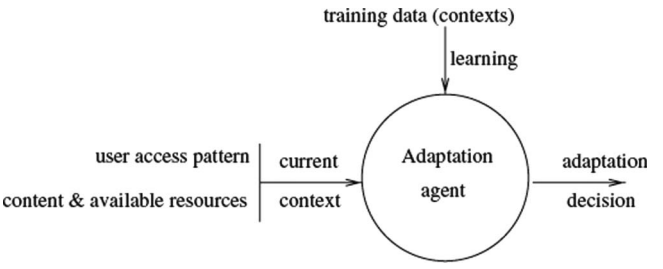


Figure 2. Adaptation agent decision by machine learning (adapted from Charvillat and Grigoraş (2007)).



distributed environments, based on rewards learnt over the time. Furthermore, approaches based on RL can operate quite well in environments with uncertainty and unexpected changes, while they can improve themselves based on received feedback. However, such approaches require modelling the alternative actions, monitoring the environment and processing received feedback from it (i.e, extra development effort (3)). Finally, the complex and gradually improving reasoning approach implies *low traceability support* (7).

### *4.5. Evaluation summary*

Throughout this section we have presented the most commonly used adaptation reasoning approaches and discussed their applicability to ubiquitous computing. The critical evaluation was based on a set of predefined requirements for dynamic adaptation reasoning in context-aware, mobile environments. This subsection summarises the advantages and disadvantages of each approach in Table 1. At this point we should mention that the presented approaches are not explicitly orthogonal. This means that reasoning techniques can be used in combination. For example, utility functions can be used in combination with RL or any other optimisation technique (see for example, Kakousis *et al.* (2008)). Furthermore, as discussed earlier, there is no one-size-fits-all solution. The decision on which approach to use depends on several factors such as the adaptation objective, the available resources, the adaptation frequency and the number of variation points.

Overall, we can say that, although action-based approaches have been extensively used as an adaptation reasoning technique (David and Ledoux 2003, Keeney and Cahill 2003), and have the significant advantage of resource efficiency, utility-based approaches are quickly gaining momentum, especially in the area of context-aware systems. Their ability for multi-dimensional evaluation and prioritisation of available variants, as well as their support for open adaptation, render this approach a very good match for ubiquitous computing. However, adaptation reasoning can be enhanced if utility functions are used in combination with other techniques from correlated fields (such as control theory, machine learning, game theory, optimisation, etc.).

## 5. Adaptation acting

Adaptation acting is responsible for applying the adaptation decisions as determined by the reasoning process. In this section, we discuss the most popular mechanisms

Table 1. Comparing adaptation reasoning approaches based on their applicability for ubiquitous computing.

| Reasoning approach/ evaluation criteria | | Action-based | Goal-based | Utility functions | Case-based | Reinforcement learning |
|---|---|---|---|---|---|---|
| 1 | Resource efficiency | H | H | M | L | L |
| 2 | Context coverage | M | L | H | L | M |
| 3 | Ease of building | L | M | L | L | L |
| 4 | Adaptation openness | L | L | H | M | M |
| 5 | Adaptation coordination | H | L | H | M | M |
| 6 | Evolvability | L | L | H | H | H |
| 7 | Traceability | H | H | L | H | L |

H(High) means that the particular approach is a very good choice with regard to a specific requirement, M(Medium) means that there is no specific reason for choosing (or not choosing) the approach for its support on a specific requirement and finally L(Low) means that the approach is not a good choice concerning a specific requirement.



found in the literature for implementing dynamic software adaptation actions. The presented mechanisms are not explicitly orthogonal and can be used in combination. In addition, the decision on which adaptation acting mechanisms to use does not depend directly on the selected reasoning approach. For the rest of this section, we present the foundations of code migration, parameterised adaptation, compositional adaptation and aspect weaving and discuss how they can be used in ubiquitous computing.

### *5.1. Code mobility*

Code mobility in distributed computing is the process of migrating or moving running program instances, codes or objects from one host to another. The set of programming languages that support code mobility are denoted in the literature as *Mobile Code Languages* (MCL) and they fall into two main categories depending whether they support *strong* or *weak* mobility. If a programming language allows execution units to move their code and execution state to a different note then it belongs to strong MCL while if it only allows the migration of the code and some initialisation data, then it is considered as weak MCL (Carzaniga *et al.* 1997). The majority of languages support weak mobility due to the costly and complex issues in defining and implementing strong mobility.

There are four possible ways of implementing code mobility depending on whether they comply with the *code pushing* or *code pulling* model (Zachariadis 2005). Code pushing refers to sending a code unit from one processing environment to another. Code pulling, on the other hand, refers to retrieving a code unit and deploying it locally. The four code mobility paradigms defined based on the above classification are:

- *Client/Server*: The client pushes a request to the server, indicating the code unit to be executed. It is mostly used in traditional distributed systems, such as in RMI and CORBA. However, the client-server model does not include actual code migration, thus it cannot be considered as a software adaptation mechanism.
- *Code On Demand* (COD): A host requests and subsequently pulls a code unit from another node. A typical example of COD is software update. Since it is

not possible to pre-load all the functionality that might be needed throughout the lifetime of a mobile device (limited resources, unpredictable context changes), COD can be used for downloading desired functionality from resources in the environment.

- *Remote evaluation*: A host pushes a particular code unit to a remote processing environment. If accepted at the destination, the unit is deployed and executed there. Common examples include SETI@home (Korpela *et al*. 2001) and Distributed.NET (Distributed.net 2009). The remote evaluation paradigm can be particularly useful for delegating resource-demanding computations from mobile devices with limited capabilities to powerful, reachable hosts.
- *Mobile agents*: A mobile agent is an autonomous code unit that is injected into the network to accomplish specific tasks on behalf of a user or an application. Adaptation in ubiquitous computing can benefit by mobile agents in situations of network unavailability or limited/costly connectivity. However, they require strong mobility, to allow for suspending at one node and resuming at another one.



Q-CAD (Capra *et al*. 2005), is a resource discovery framework that enables pervasive computing applications to discover and select the resources best satisfying the user needs, based on the current context situation. The resource selection is based on *Utility Functions* (see Section 4.3) that select those resources that maximise the application's utility. Code mobility techniques are used for enabling adaptation. In particular, remote evaluation is used for deploying utility functions on remote hosts while CoD is used for downloading any remote component needed to perform adaptation.

Overall, code mobility provides several benefits for software adaptation in ubiquitous computing. Zachariadis *et al*. (2003) suggested that techniques from the area of code mobility may contribute in advancing ubiquitous computing applications, by allowing for a greater degree of adaptability, dynamicity and reaction to context. In particular, applications deployed on mobile devices can dynamically acquire new functionality and services that have not been envisioned at design time. In addition, code mobility facilitates resource efficiency since mobile devices can use available resources from nearby powerful hosts. However as presented in Zachariadis (2005), some heterogeneity, binding and security issues may arise when using code mobility. For example, the transferred code is usually compiled for a specific architecture and it might not be compatible with some hardware platforms or operating systems.

### 5.2. *Parameter adaptation*

Parameter adaptation is used for denoting adaptation techniques that affect the system's behaviour through the modification of specific program variables. A well-known example of parameter adaptation comes from the area of networks and the way the TCP protocol adjusts its behaviour in response to network congestion (Hiltunen and Schlichting 1996, Kurose and Ross 2002). In the scope of ubiquitous computing, parameter adaptation is used for modifying the extra-functional properties of a system that are affected by context changes. For example, the image render of a mobile device may change the quality of the presented images depending on the availability of bandwidth and memory. Thus, adjusting specific parameters may alter the application's behaviour to improve its fitness to the current context.

However, in highly dynamic systems where unanticipated context changes occur rather frequently, identifying all the possible context states and calculating values for the extra-functional properties in advance is not always feasible. Parameter adaptation is a cheap (in terms of complexity and implementation effort) and lightweight dynamic adaptation solution, when compared to similar techniques (such as code migration). It is a proven technique as it has been recently used in multiple context-aware systems (Fickas *et al*. 1997, Salber *et al*. 1999, Flinn *et al*. 2001, Kortuem *et al*. 2001, Minar *et al*. 2000, Sousa and Garlan 2002). However, it is not considered sufficient to be used as a holistic adaptation solution for ubiquitous applications. Most importantly parameterised adaptation cannot adopt software components and algorithms left unimplemented during the original design of the system. Moreover, for applications highly exposed to frequent context changes, the numerous possible values of discrete configurable parameters may lead to a combinatorial explosion (Brataas *et al*. 2007b). This raises the need for extra adaptation mechanisms which reduce the number of parameter values under consideration (e.g. delegation of parameter adaptation to a lower level, where the number of possible reconfigurations is reduced, and discretisation of large and continuous value ranges to a small set of interesting values).



### *5.3. Compositional adaptation*

Compositional adaptation goes far beyond the simple code tuning, programmed during design time, and allows exchange of algorithmic and structural parts of the system in order to improve a program's fit to its current environment. Dynamic recomposition copes with situations where unanticipated conditions or requirements occur and consequently new adaptation functionality is needed. Because of the great potential offered in ubiquitous computing by compositional adaptation, several research groups have contributed to the establishment of a well-defined research topic. Here, we review main technologies behind compositional adaptation and subsequently discuss how available approaches can be classified with regard to *when* and *where* compositional adaptation takes place (Aksit and Choukair 2003, McKinley *et al*. 2005).

Compositional adaptation is build upon fundamental principles of synchronous software engineering such as *Separation of Concerns* (SoC) (Hoffman and Weiss 2001), *component-based design* (Szyperski 1997), *computational reflection* (Maes 1987) and *architectural models* (Floch *et al*. 2006). SoC is a development principle (and process) where the business logic (i.e. the functional part) of an application is developed separately from the cross-cutting concerns (e.g., QoS, security and fault tolerance). It increases stability, maintainability, reusability and extensibility while at the same time it simplifies development. Among the several development techniques that support SoC (generic programming, constraint languages, feature-oriented development, etc) *Aspect Oriented Programming* (AOP) appears to be the most commonly used approach (McKinley *et al*. 2005) and it is further discussed in Section 5.4.

Compositional adaptation belongs to the traditional component-based paradigm where different application concerns are developed into individual components which can be reused separately to form new applications. *Component frameworks*, as defined by Szyperski (1997), regulate how components interact with each other and

with their environment. They usually implement protocols for connecting participating components and enforce policies for how component instances perform specific tasks.

Design methodologies that clearly separate and externalise adaptation logic from the application implementation satisfy a fundamental requirement for compositional adaptation. Middleware-based solutions are the most common for encapsulating and reusing common adaptation functionality. As a reusable software layer lying between the operating system and the application layer, middleware enables the execution of collaborating components in a heterogeneous and distributed execution environment (Hallsteinsen *et al*. 2005, da Rocha and Endler 2006) while it avoids intertwining the adaptation and application logics. On the contrary, alternative programming language solutions do not separate adaptation concerns and often introduce intolerable complexity. Typical middleware examples are CORBA, COM + and J2EE. Several works (Hallsteinsen *et al*. 2005, Gjørven *et al*. 2006, Geihs *et al*. 2009a) on self-adaptation use middleware-based approaches in order to encapsulate the adaptive behaviour of their systems.

Besides middleware, *computational reflection* and *architectural models* can also support externalised adaptation reasoning. The former refers to the ability of a system to reason about itself (introspection) and act upon this information (intercession). Reflection is indirectly available through programming language features or directly implemented in middleware platforms (Capra *et al*. 2003,Gjørven *et al*. 2006). Architectural models have been used traditionally in *Model Driven Architectures* (MDAs) and *Architectural Description Languages* (ADLs) (Pandey 2010) for representing architectural design and implementation of software systems in static models (Geihs *et al*. 2009b). However, the current research efforts focus on making such architectural models available at runtime. The target is to utilise architectural models as an external (and platform independent) adaptation mechanism where developers are allowed to represent their systems as compositions of components and monitor how components interact, what are their properties and how they adapt, throughout the whole lifecycle of the system. Floch *et al*. (2006) and Geihs *et al*. (2009a), describe how software variability can be encoded into an architectural model, so an underlying middleware can automatically select, among feasible application compositions, the one that best fits to the current context. Variability is achieved by providing different alternative realisations of the component types constituting an application (i.e. provide the same functionality with different extra-functional behaviour). An underlying middleware, through a planning process, generates and evaluates (using utility functions) every possible application variant. *Utility functions* (mathematical artefacts and also a well-known adaptation reasoning approach, see Section 4.3) assign a scalar value to alternative application variants, depending on their suitability for the current context. As a conclusion, the most powerful compositional adaptation approaches are supported by middleware, computational reflection and runtime architectural models. Each one of the above factors can contribute positively in leveraging the advantages of compositional adaptation.

#### *5.3.1. When to compose*

In respect to when compositional adaptation occurs, we distinguish between two main categories. *Static* composition refers to composition mechanisms that take

place either at design, compile or load-time. On the other hand, *dynamic* composition includes methods that can be applied at runtime. Generally speaking, later adaptation solutions are more powerful but they also increase complexity and may introduce inconsistency and abnormal adaptation behaviour. Although static composition may introduce a level of adaptability, no mobile self-adaptation solution would be feasible without dynamic composition. Composition mechanisms of this category allow replacing and extending structural and algorithmic units of a program, without halting or restarting its execution. We differentiate between two types of dynamic composition: *Tunable software*, that only permits changes to the extra-functional aspects of an application, and *mutable software* which allows recomposition, which may also alter the business logic of a program. Systems targeting pervasive and mobile environments traditionally belong to the tunable software classification.



### 5.3.2. *Where to compose*

This composition dimension refers to where in the system the adaptive code is to be inserted. Insertion possibilities include the application code itself or one of the middleware layers. There is also the possibility of extending the operating system to incorporate adaptation behaviour. However, we do not consider such adaptation solutions since they are platform-dependent and do not satisfy the portability requirement of ubiquitous computing. Conversely, higher middleware levels effectively support adaptability since they are platform independent and transparent in respect to the application program. On the other hand, such approaches only support applications developed against a specific middleware platform. Approaches that implement compositional adaptation in the application program itself aim to overcome this limitation. They achieve compositional adaptation either by implementing all or part of the application code using languages that directly support dynamic recomposition (such as CLOS, Python, Adaptive Java, etc) or by weaving the adaptive code into the functional code. Although such methods maximise portability, they intertwine adaptation and application logics and complicate the development process.

An alternative solution, gaining momentum for dynamic adaptation in ubiquitous computing, is to model adaptability at the application level through adaptation models and encapsulate adaptation services into the high middleware layers. Such approaches separate adaptation and business logic, facilitate reusability and accommodate application development. Model-based approaches that support dynamic software adaptation are based on a model of the system and its context. The model is regularly developed at design time and utilised throughout the whole lifecycle of the system. Modelling approaches include, among others, *Architectural Models* and ADLs (Garlan *et al*. 2004) as well as *formal methods* (Laddaga *et al*. 2003) and *domain specific models* (Gabor *et al*. 2003).

However, existing approaches for model-based adaptation of context-aware systems mainly focus on Software Architecture models and ADLs. In Section 5.3, we already discussed how architectural models in MADAM (Floch *et al*. 2006, Geihs *et al*. 2009a) have been utilised at runtime for adapting context-aware applications. The IST-MUSIC (2009) project builds on the same model-driven development approach for extending adaptation capabilities of the middleware to support service-oriented adaptation as well. Henricksen and Indulska (2005) presented a set of

conceptual models designed to facilitate the development of context-aware applications by introducing greater structure and improved opportunities for tool support into the software engineering process. Finally, Sindico and Grassi (2009) presented initial results of a model driven development framework for context awareness. Its core element consists of a domain specific modelling language defined as a UML extension. The language can be used to enrich a UML model of an application with context elements. The goal is to provide a model transformation aimed at generating executable code for context-aware applications.

#### *5.3.3. Should we compose?*

Compositional adaptation is a more generic approach on self-adaptation that depends on the implementation language of the self-adaptive application and affects the whole adaptation loop. Based on the numerous research works that have been conducted around compositional adaptation (Hallsteinsen *et al*. 2005, Floch *et al*. 2006, Gjørven *et al*. 2006, Geihs *et al*. 2009a), we claim that it is a powerful solution, mainly for two reasons. At the reasoning (or decision-making) level, it provides more flexibility since the reasoning logic is applied on smaller and more autonomous software artifacts (i.e. components). At the adaptation acting level, dynamic software reconfigurations are easier to implement while at the same time are more powerful since they enable algorithmic and structural changes.

However, certain types of adaptations are better covered by alternative solutions. For example, specific adaptation actions, related to cross-cutting concerns such as, security and QoS, are better handled by aspect weaving which is covered in the next section. Moreover, for simple application behaviour modifications, compositional adaptation should be used in combination with parameterised adaptation which avoids the overheads and scalability problems (see Section 6.2) that may occur with a more sophisticated adaptation approach. Finally, the large scale changes permitted by compositional adaptation often require an underlying framework that realises dynamic software reconfigurations and controls the adaptation process in general (Geihs *et al*. 2009a).

### *5.4. Aspect weaving*

AOP was developed in order to enhance software modularisation by enforcing SoC. In particular, AOP facilitates the individual implementation of cross-cutting concerns such as security, QoS, fault tolerance, etc. The implementation of such a concern is called *aspect* and specialised descriptions, called *pointcuts*, indicate the set of *join points* where aspects execute. Join points represent runtime conditions that arise during program execution or locations in the structure of the source code. The occurrence of such a location or condition triggers the execution of aspect behaviour. The process of inserting or removing aspects into join points is called aspect weaving and may occur at run-, load- or compile-time (Kiczales 1996, McKinley *et al*. 2005, Grace *et al*. 2008, Rouvoy *et al*. 2008a).

Almost immediately, AOP was correlated with the component-based design paradigm and compositional adaptation. The goal was to enable adaptation of cross-cutting concerns in applications assembled from complex compositions of local components and remote services that often deal with non-functional concerns. This initiative was supported by the development of multiple aspect-component

frameworks, such as JAC (Pawlak *et al*. 2004), Prose (Popovici *et al*. 2002) and JBoss AOP (JBossAOP 2009) that promoted the *Aspect Oriented (AO) composition*. In AO composition, aspects define behaviour and composition logic describing where and when this behaviour is executed. Component frameworks enable aspect-behaviour reuse through the separation of aspect behaviour from composition logic.

According to Grace *et al*. (2008), traditional aspect-component frameworks support only coarse-grained adaptation since the entire aspect (i.e. behaviour and composition logic) must be added or removed from the application. AspectOpenCOM (Grace *et al*. 2008) aims to provide a finer-grained adaptation where the elements that compose an aspect can be reconfigured individually. In addition, Rouvoy *et al*. (2008a), suggest the integration of AOP principles in a planning-based middleware to manage adaptation cross-cutting concerns. On one hand the middleware tries to maximise the user's satisfaction with the minimum adaptation and resource cost and on the other hand, the adoption of aspects in combination with the component-based design, enables the identification and isolated implementation of cross-cutting adaptation concerns.

Although combining aspect weaving and compositional adaptation has been proven a viable approach with several advantages (Rouvoy *et al*. 2008a), there are is still some criticism related to AOP adoption issues. Programmers need to learn a new programming language and understand what is actually happening in the aspects code in order to prevent errors. Currently, there is a lack of visualisation and other design tools that could assist aspect programming. Finally, given the power of AOP, a logical mistake in expressing a cross-cutting concern may lead to a widespread program failure.

### 5.5. *Summary*

The existing literature already provides a number of extensive evaluations of software adaptation acting mechanisms (McKinley *et al*. 2005, Kell 2008). In this section, we presented a subset of the possible adaptation techniques, which are better fitted for mobile and ubiquitous computing. Summarising the results that were discussed in this section, we argue that compositional adaptation provides the flexibility needed for adapting applications in highly dynamic environments such as in ubiquitous and mobile computing. However, it can be argued that a complete adaptation acting approach should combine more than one acting technique, depending on the domain that it is applied in. First, parameterisation can complement compositional adaptation in situations where the desired behaviour is possible, simply by fine tuning specific application variables. Aspect weaving, on the other hand, improves modularity and enables handling cross-cutting concerns more efficiently. Finally, applying techniques from code mobility in some cases can improve the adaptation process by benefitting from available nearby resources or by deploying mobile agents in situations of network unavailability or limited/costly connectivity.

## 6. Discussion

Pervasive systems, as an evolution of distributed systems (Satyanarayanan 2001), introduce new challenges in the development of software: increased mobility, scarce

resources, limited connectivity, etc. These characteristics heavily affect the adaptation process and the way the developers decide which adaptation mechanism or reasoning approach to follow. In this section, we discuss some of the key challenges (Section 6.1) and open research issues (Section 6.2) in adaptation for ubiquitous computing. The list is by no means exhaustive because dynamic software adaptation in mobile environments is a multi-dimensional aspect, intermixing concerns from several domains with some of them being outside the scope of this work (e.g. security, testing, etc). Finally, we provide some directions for future research work (Section 6.3).

## *6.1. Challenges*

### *6.1.1. Distributed adaptation*

Distribution plays a special role in dynamic adaptation. First of all the adaptation reasoner should take into consideration available resources, not only from local but from distributed devices as well. Then, an adaptation in one node may trigger adaptations to other nodes, while the discovery of new context sensors and service providers increases the choices in alternative component implementations and hence, the adaptation scalability problem as well (see Section 6.2). Furthermore, security and transactional guarantees need to be strengthened in an unreliable distributed environment.

Generally speaking, distributed adaptation decisions are classified to *centralised* and *decentralised*. The former refers to approaches where a master node controls the adaptation reasoning and invokes adaptation services on slave nodes. The latter implies that distributed nodes are able to make local adaptation decisions and collaborate with other nodes in order to achieve global adaptation. The research community has not reached an agreement yet in favour of centralised or decentralised adaptation reasoning. Centralised adaptation management and reasoning is much simpler and allows for adaptation reasoning based on complete and locally available context information. On the contrary, decentralised approaches introduce complexity and require negotiation among distributed nodes, in order to achieve convergence of the adaptation loops. In addition, adaptation in decentralised approaches can only produce approximations of the optimal solution because reasoning is based on partial context information. On the other hand, in decentralised adaptation there is no need for gathering context information to a central node. This prevents time-consuming context updates and enables monitoring of context changes during the adaptation process.

Scholz and Rouvoy (2008) suggested a divide and conquer-based technique for organising component based adaptation in distributed environments. They propose a decentralised and distributed solution for splitting and adapting applications into smaller units (parts) and collections of application units (packs). The idea is to use the minimum amount of resources in order to partially adapt applications, leaving untouched application components that have not been affected by the particular context change that triggered the adaptation. Although a promising approach, it is still in progress and it has not been validated whether it can drastically tackle scalability problems without introducing extra overhead and complexity at either design-time or runtime.

### 6.1.2. *Adaptation with components and services*

In ubiquitous computing environments where network connections appear and disappear unpredictably and devices constantly move, preserving the availability and QoS of the required and provided services of a self-adaptive application becomes a challenging concern. In response to this, the emergence of Service Oriented Architectures (SOA) (Erl 2005) can play an important role in the development of self-adaptive applications. Services are reusable and composable entities that can be discovered and exploited dynamically without prior knowledge of their underlying platform implementation.

SOA is considered an evolution of component-based architectures where services are dynamically discovered, bound and exchanged. In addition to these, SOA incorporates a business model where software is provided as a ready-to-use functionality from a service provider to a service consumer probably after agreeing to a service contract that defines the capabilities and QoS properties of a service. On the contrary, component-based systems require that the service consumer provides the necessary infrastructure for instantiating and executing a software component implemented by the software provider.

Central to the SOA paradigm is the notion of *Service Level Agreement* (SLA) (Heiko 2007), a part of the service contract that describes the expected behaviour of the service in terms of performance, reliability and other QoS properties. SLA is a negotiable agreement between service providers and consumers. The former are responsible for monitoring the service quality and possibly adapting resource distribution to avoid an SLA violation. Service consumers may also monitor the provided QoS to avoid blindly trusting the provider.

With the proliferation of service oriented computing, mobile devices that move about inside ubiquitous environments discover and interact with a dynamic service landscape. Therefore, additional requirements to self-adaptation arise and new adaptation mechanisms that utilise service-oriented context information are needed. State-of-the-art research projects have already provided initial results for accommodating service-oriented context changes for adaptation of mobile applications (Papakos *et al.* 2009, Romero *et al.* 2009). For example, Rouvoy *et al.* (2009) describe how in IST-MUSIC (2009), a middleware that provides a QoS-driven generic planning framework for self-adaptive mobile applications, has been extended to seamlessly support both component-based and service-based configurations. The framework has been enhanced to adapt to changes in a landscape of ubiquitous remote services that dynamically come and go, and where the offered service qualities vary. The planning middleware discovers remote services and evaluates them as alternative providers for the functionalities required by an application. The framework exploits these changes to maximise the overall utility of applications.

Based on the above, we argue that a modern self-adaptive system can benefit in terms of usability, reliability and availability if adaptation facilities in the business service landscape are also considered. In addition, SOA is suitable for modern enterprise architecture since it provides flexibility and adaptability through loosely-coupled integration by utilising system implementation technology such as web services paradigms (Kong *et al.* 2009). However, extending traditional component frameworks to incorporate services is not a straightforward process. Additional development effort is needed for enabling discovery of new services, monitoring and detecting changes in service availability and QoS properties. Moreover, considering

SLA negotiations, violations and service denials during adaptation planning requires in-depth investigation. Finally, in cases where inter-organisation services are also considered, we might encounter interoperability problems between providers using different adaptation decision methods, while the already exponential problem of adaptation scalability (Section 6.2) is increased due to the plethora of alternative service and/or component implementations.

### 6.1.3. *Artificial intelligence-based reasoning*

As shown in Figure 1, adaptation in ubiquitous computing can be seen as a closed loop, comprised of three consecutive phases: *context sensing and processing, adaptation reasoning* and *adaptation acting*. This is very similar to the *Sense-Plan-Act* (SPA) architecture used in robotics (Brooks 1986), and also extensively studied in the field of AI (Tesauro and Kephart 2004). Planning in AI is an active research topic, fuelled by the need for planning the behaviour of autonomous agents. Although AI-based techniques are usually resource intensive, we believe that investigating the possibility of applying main concepts of AI planning techniques to ubiquitous adaptation reasoning is a promising direction.

In AI, planning is the process of generating a sequence of actions whose execution adapts the system to a desired state (Russell and Norvig 2002). We distinguish between classical and non-classical planning. The former refers to planning that occurs in static environments which are fully observable, deterministic, finite and discrete. Conversely, non-classical planning refers to more sophisticated approaches, suitable for dynamic and stochastic environments much closer to pervasive ones. Gradually relaxing restrictions, imposed in classical planning, led to several non-classical techniques, much more expressive and powerful, but also in some cases more complex. For example, *Temporal planning* (Smith and Weld 1999) extends classical planning by allowing actions to have different durations and also to overlap each other. As opposed to classical planners where the only objective is finding the shortest plan, in temporal planning the user is allowed to define multiple objectives (e.g. better performance, improved quality or a combination of them). Although temporal planning is a much more expressive and realistic approach than classical methods, it is often too hard to implement or too domain-specific to be used as a generic adaptation solution.

Beyond the above classification, Majercik and Littman (2003) provide a multi-level categorisation of AI planners (*Deterministic, Probabilistic* and *non-probabilistic* planners, *contingent* and *conditional* planners). For each possible classification several planners have been proposed, indicating that planning in AI is very flexible, allowing for the development of planners customised to the domain of interest.

In addition to the aforementioned AI techniques, emerging approaches from AI have also been proposed for enabling adaptation reasoning. As an example we cite an approach from Ruiz *et al.* (2004) where a combination of *genetic* and *rule-induction* algorithms compose the adaptive behaviour of multimedia applications. In particular, a genetic algorithm is used (after several experimentations and simulations) to decide when (i.e. under which network conditions) a multimedia application should adapt in order to maximise user satisfaction. On the other hand, a rule induction algorithm, called SLIPPER (Cohen and Singer 1999), generates formal rules that represent the user's perception of QoS. Of course, such a

representation requires extensive learning examples that have been evaluated and scored by real users.

Although many concepts from the area of AI can form the basis for sophisticated software adaptation in ubiquitous computing, we have not seen yet any application of adaptation reasoning approach, fully based on AI. We believe that the main reasons for this are the low and often unpredictable performance of such approaches, their resource demands, as well as the inherent need for modelling the problem. We consider the incorporation of AI techniques in dynamic compositional adaptation as an open research problem.

### 6.2. *Compositional adaptation scalability issues*

Designing and implementing the adaptation decision process is probably the most crucial and at the same time the most complex part of component-based, self-adaptive systems. We have already discussed possible adaptation reasoning mechanisms and explained how variability can be modelled in component-based systems through architectural models. We have assumed, however, the existence of an adaptation planning framework responsible for modelling different variation points and capable of dynamically searching and selecting the most suitable component composition for the current context. Most commonly, the evaluation of alternative composition is implemented using component-wise utility functions that assess the suitability of alternative application variants to the current context, using property predictors which provide necessary information about QoS-properties and resource requirements (Brataas *et al*. 2007a).

The planning process, which can be triggered either at runtime or during the deployment of the application, proceeds in two phases. The first one corresponds to the discovery and assembly of possible application variants as they can be generated by resolving variation points, defined in the component framework. During the second phase, the configuration steps are decided in a way that ensures timely and safely transformation to the new configuration.

Such a variability model, where application variants increase exponentially with the number of variation points, introduces huge limitations and scalability issues for resource constrained devices. For example, if an application is composed of five components and there are five alternative implementations for each one of them, the application has $5^5 = 3125$ variants. The problem becomes even worse when more than one application is allowed to adapt simultaneously. Imagine, for example, having two applications with five components each; in this case the number of variants increases dramatically to a number approaching 10 millions. Thus, in that case we would need to (re)evaluate millions of variants and (re)calculate the same number of utility functions along with their corresponding property predictors.

Dealing with adaptation scalability problems, requires effective ways for handling large amounts of variants, for instance by reducing the number of variants that need to be considered. In addition, some inexplicit factors may also affect adaptation performance. For example, how often should we inspect context changes and trigger adaptation? What is considered as reasonable adaptation time, and for how long is the user willing to wait? Is the optimal adaptation variant always required, even if reasoning for it causes performance tradeoffs?

Several heuristics have been proposed for dealing with scalability problems imposed by planning frameworks. In Aura (Sousa *et al*. 2006), for example, the

authors have managed to restrict the search space by defining two-part utility functions: the functional preferences and the QoS properties. The overall utility is calculated as their normalised product (i.e. both utility parts take values between 0 and 1 and hence their product is also restricted in the range [0 ... 1]). Functional preferences are more static and already known when evaluating QoS properties. This is how they introduce an upper bound for the utility score. Consider for example two services a and b with two possible variants each. The possible configurations are: (a1, b1), (a1, b2), (a2, b1) and (a2, b2). Assuming that the already known functional utilities are 0.8, 0.6, 0.4 and 0.2 respectively then, if during QoS-properties evaluation the combined utility of (a1, b1) is found to be higher than 0.6, there is no further need for calculating the remaining QoS utilities, since the overall utility cannot be higher than 0.6. A similar two-step approach is also adopted by the Q-CAD (Capra *et al.* 2005) project with the main difference being that both functional and non-functional requirements are considered during the first step.

Finally, Khan *et al.* (2008), describe how to restrict the number of alternative variants by imposing architectural constraints between components. For example, choosing one component may only be justified if a second one is chosen as well. Conversely, two components might be mutually exclusive and only one of them might participate in a composition plan.

As a conclusion, we believe that the adaptation scalability problem in resource-constrained devices is a multi-dimensional open problem that needs to be considered early in the life-cycle of adaptable systems. Available heuristics can only provide limited solutions and often introduces extra complexity.



### *6.3. Future directions*

#### *6.3.1. Alleviating developers*

In Section 4, we presented an extensive evaluation of available adaptation reasoning mechanisms. From our discussion, and the table in Section 4.5, it can be inferred that most of the approaches require considerably high development effort from the perspective of application developers. Context-aware, self-adaptive applications demand from their developers to make difficult and crucial decisions at design time that affect the runtime properties of the system such as its performance and evolving behaviour (e.g. due to the discovery of new components and services, or changes to the user requirements).

Model-based approaches are supported by the plethora of existing modelling languages and tools (Object Management Group 2009) that can ease the development process. Using modelling tools, application developers can define through architectural models the variation points that will enable dynamic adaptation. Such architectural models can be enriched with property descriptions and evaluation mechanisms that enable the adaptation manager to decide on the best configuration at runtime (Floch *et al.* 2006). Furthermore, model transformation approaches (Czarnecki and Helsen 2003) and tools can be used for generating considerable amounts of code directly from architectural models.

In addition, Context-Oriented Programming (COP), a relatively novel programming paradigm, aims to reduce the complexity of the development of context-aware applications. In COP context information is considered as a first-class construct of a

programming language, much in the same way that variables, classes, and functions form the first-class constructs of many modern languages (Keays and Rakotonirainy 2003, Hirschfeld *et al*. 2008). COP reduces development effort since reasoning about context is no longer the responsibility of the developer.

Although the aforementioned techniques can alleviate the development process, they either require modelling the application's adaptation behaviour or familiarising with a different programming paradigm. We envision developer-friendly and more generic approaches and tool support that will enable developers to provide reusable adaptation functionality while minimising the required effort.

### *6.3.2. Self-adaptive adaptation reasoning and adaptation balance*

In ubiquitous computing, where hand-held devices are exposed to constant mobility, it is more than possible for the initially selected adaptation strategies to become obsolete due to significant context changes. If, for example, the user goals and preferences, or the system requirements change during execution time, the adaptation logic should be reconsidered. In such circumstances, a truly adaptable system should be able to reason about and reconfigure the adaptation process itself. This is a rather complex task where adaptation process should be constructed in a way that allows adapting itself as if it were a regular adaptable application. For example in the case of middleware-based adaptation, the underlying middleware should be able to adapt itself without suspending the currently executing adaptations. Although some existing approaches (Gjørven *et al*. 2006, Geihs *et al*. 2009a, IST-MUSIC 2009) have attempted to provide such a facility, their results are still under investigation and have not been fully validated.

A significant benefit from having self-adaptive adaptation logic is for achieving an adaptation balance between application autonomy and user control. A main objective of pervasive computing is to minimise the user distraction and hence enable transparent application adaptation to the end user's preferences. One could argue that maximising the application's autonomy will eventually enable the desired *calm technology* as it was envisioned by Weiser and Brown (1997).

However, as van der Heijden (2003) explains, taking the control from the user and giving it to the application increases the user's anxiety: '*the personal discomfort that the end user relates with the use or the behaviour of the system*.' Moreover, increased autonomy in combination with black-box decision-mechanisms are very likely to lead to unexpected and unwanted behaviours. Barkhuus *et al*. (2003) noted that users are willing to accept loss of control to some extent, as long as the usefulness of an application overcomes the cost of limited control. But how do we define the threshold between usefulness and anxiety? Is it the same for every user? Hardian *et al*. (2006) stressed the need for finding a balance between autonomy and user control in context-aware systems, and also revealed the lack of existing work towards this direction. Hardian, argues that a good balance can be achieved by selectively revealing context information to the end users which can then judge its accuracy.

Adaptable adaptation reasoning is needed for achieving the so called adaptation balance. For example, in Kakousis *et al*. (2008) we suggested a utility function-based approach for adjusting the adaptation reasoning process itself. Weighted utility functions have been used for specifying the adaptive behaviour of component-based, context-aware systems. This approach leverages potential user feedback for adjusting utility weights in order to optimise adaptation reasoning according to the user preferences.

Other existing approaches are either primitive or based on end-user programming which enables adding functionality that has not been anticipated by the system designer. Unfortunately, the latter is not appropriate for every application domain, especially for those that involve complex adaptation tasks. Based on the above, we stress the need for more effective solutions that will allow dynamic optimisation of the adaptive behaviour through modification of the actual adaptation decision process.

## 7. Conclusions

In this work we have reviewed the state-of-the-art in software adaptation with emphasis on applications in mobile and ubiquitous computing. Adaptive mobile systems are characterised by a major constraint that significantly affects the way that they should be developed: while the context changes are more frequent than in most other computing environments, the resources are extremely scarce. This leads to the necessity for viable software development solutions that facilitate both resource preservation and maximum adaptation capabilities. Naturally, the development effort for applications featuring such sophisticated behaviour is drastically increased and the complexity grows even further as the boundary between cyberspace and real world fades away. This is also true in the case of enterprises transitioning to the ubiquitous computing era, where we expect a variety of heterogeneous systems to communicate in real-time through the internet and provide their functionality in a decentralised and autonomously adaptive manner. It is argued that self-adaptive enterprise systems will be more appropriate for handling the increased amount of data and information from distributed sources (Bechini *et al*. 2008), as well as for adjusting the human–computer interfacing (Barkhuus *et al*. 2003, Schmidt 2000). Furthermore, we argue that enterprise systems capable of adjusting their behaviour at run time, accomplishing their own business rules independently, and if necessary collaborating with decentralised systems (Kong *et al*. 2009) will be increasingly desired and successful. Such an increased level of process automation leads to reduced cost for development and maintenance, since less human intervention is required and human errors are reduced (Strassner and Schoch 2003).

In this article we have surveyed runtime adaptation reasoning approaches and acting mechanisms, and presented a classification and critical evaluation of them. From our discussion, it can be inferred that utility-based reasoning approaches are gaining momentum in the area of context-aware systems. This is mainly due to their ability of multi-dimensional evaluation and prioritisation of available variants, as well as their support for open adaptation. In addition case-based approaches and RL can also contribute to dynamic software adaptation mainly due to their ability to learn from past experience and improve their decision mechanism. However, we argue that there is no one-size-fits-all solution. The decision on which approach to use depends on several factors such as the adaptation objective, the available resources, the adaptation frequency and the number of variation points.

With regard to the adaptation acting mechanisms, there is a general consensus that component-based approaches supported by middleware, reflection and possibly architectural models, provide the prerequisites for dynamic, context- and QoS-aware adaptation (Capra *et al*. 2003, David and Ledoux 2003, Alia *et al*. 2006, Zachariadis and Mascolo 2006). Indeed, some of the most recent projects in the area, such as QuA (Gjørven *et al*. 2006), MADAM (Geihs *et al*. 2009a) and MUSIC (IST-MUSIC 2009) have exploited the power of the aforementioned technologies in order to satisfy their adaptation requirements. In addition, utilising powerful component frameworks is

often beneficial in the development of self-adaptive systems since they support hardware, networking and operating system interoperability and allow hosting of services by different providers and at different security levels (Gu *et al*. 2004).

Finally, in Section 6 we have discussed main research challenges in enabling dynamic software adaptation in ubiquitous computing. In particular, we indicated the difficulties in enabling decentralised distributed adaptation and revealed the necessity for incorporating the available functionality from the service landscape to the adaptation reasoning process. We discussed the similarities between AI planning and dynamic software adaptation and described how adaptation scalability issues may arise in compositional adaptation. Finally, we indicated the need for providing tools and methodologies to alleviate developers and we argued that self-adaptive reasoning itself should be open to adaptation and evolution.

We close this article by observing that while the progress in software adaptation has gone a long way, there is still a great need for further improvement. This article aimed not only at surveying the state-of-the-art in software adaptation, but also at contextualising it in the scope of mobile and ubiquitous computing. Finally, it also discusses a few open research paths that appear promising towards bringing software adaptation into mainstream software engineering.



## References

Aksit, M. and Choukair, Z., 2003. Dynamic, adaptive and reconfigurable systems overview and prospective vision. *In*: *Distributed computing systems workshops, 2003. Proceedings of 23rd international conference*, ICDCS, 19–22 May 2003, Providence, RI, 84–89.

Alia, M., *et al*., 2006. A component-based planning framework for adaptive systems. *In*: *The 8th international symposium on distributed objects and applications (DOA)*. Berlin/ Heidelberg: Springer, 1686–1704.

Baldauf, M., Dustdar, S., and Rosenberg, F., 2007. A survey on context-aware systems. *International Journal of Ad Hoc and Ubiquitous Computing*, 2 (4), 263–277.

Barkhuus, L., *et al*., 2003. Is context-aware computing taking control away from the user? Three levels of interactivity examined. *In*: *Proceedings of Ubicomp 2003*. Springer, LNCS, 149–156.

Bechini, A., *et al*., 2008. Patterns and technologies for enabling supply chain traceability through collaborative e-business. *Information Software Technology*, 50 (4), 342–359.

Becker, S., *et al*., 2006. Towards an engineering approach to component adaptation. *Architecting Systems with Trustworthy Components*, Volume 3938/2006, 193–215.

Bell, G. and Dourish, P., 2007. Yesterday's tomorrows: notes on ubiquitous computing's dominant vision. *Personal Ubiquitous Computing*, 11 (2), 133–143.

Bolchini, C., *et al*., 2009. And what can context do for data? *Communications of the ACM*, 52 (11), 136–140.

Brataas, G., *et al*., 2007a. A basis for performance property prediction of ubiquitous self-adapting systems. *In*: *ESSPE '07: International workshop on engineering of software services for pervasive environments*. New York, NY: ACM, 59–63.

Brataas, G., *et al*., 2007b. Scalability of decision models for dynamic product lines. *In: SPLC (2)*. Tokyo, Japan: Kindai Kagaku Sha Co. Ltd., 23–32.

Brooks, R., 1986. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, 2 (1), 14–23.

Canal, C., Murillo, J.M., and Poizat, P., 2004. Coordination and adaptation techniques for software entities. *In*: *Object-oriented technology. ECOOP 2004 workshop reader*, chapter *Coordination and adaptation techniques for software entities*. Oslo, Norway: Springer, 133–147.

Canal, C., Poizat, P., and Salau"n, G., 2008. Model-based adaptation of behavioral mismatching components. *IEEE Transactions on Software Engineering*, 34 (4), 546–563.

Capra, L., Emmerich, W., and Mascolo, C., 2001. Reflective middleware solutions for context-aware applications. *In*: *REFLECTION '01: Proceedings of the 3rd international conference on metalevel architectures and separation of crosscutting concerns*. London, UK: Springer-Verlag, 126–133.

Capra, L., Emmerich, W., and Mascolo, C., 2002. A micro-economic approach to conflict resolution in mobile computing. *SIGSOFT Software Engineering Notes*, 27 (6), 31–40.
Capra, L., Emmerich, W., and Mascolo, C., 2003. CARISMA: context-aware reflective middleware system for mobile applications. *IEEE Transactions on Software Engineering*, 29, 929–945.
Capra, L., Zachariadis, S., and Mascolo, C., 2005. Q-CAD: QoS and context aware discovery framework for mobile systems. *In*: *Pervasive Services, 2005. ICPS '05. Proceedings of international conference,* July. Santorini, Greece: IEEE Computer Society, 453–456.
Carzaniga, A., Picco, G., and Vigna, G., 1997. Designing distributed applications with mobile code paradigms. *In*: *Software engineering, 1997. Proceedings of the 1997 (19th) international conference,* May. Boston, MA: IEEE Computer Society, 22–32.
Charvillat, V. and Grigoraş, R., 2007. Reinforcement learning for dynamic multimedia adaptation. *Journal of Network and Computer Applications*, 30 (3), 1034–1058.
Chen, G. and Kotz, D., 2000. *A survey of context-aware mobile computing research*. Technical report. Hanover, NH: Dartmouth College.
Cohen, W.W. and Singer, Y., 1999. A simple, fast, and effective rule learner. *In*: *AAAI '99/IAAI '99: Proceedings of the 16th national conference on artificial intelligence and the 11th innovative applications of artificial intelligence conference innovative applications of artificial intelligence*. Menlo Park, CA: American Association for Artificial Intelligence, 335–342.
Conan, D., Rouvoy, R., and Seinturier, L., 2007. Scalable processing of context information with COSMOS. *In*: *Proceedings of the 7th IFIP international conference on distributed applications and interoperable systems (DAIS'07)*, Vol. 4531. Paphos, Cyprus: Springer Verlag, 210–224.
Czarnecki, K. and Helsen, S., 2003. *Classification of model transformation approaches* [online]. Available from: http://gsd.uwaterloo.ca/ [Accessed 24 December 2009].
da Rocha, R.C.A. and Endler, M., 2006. Middleware: context management in heterogeneous, evolving ubiquitous environments. *IEEE Distributed Systems Online*, 7 (4), 1.
David, P.C. and Ledoux, T., 2003. Towards a framework for selfadaptive component-based applications. *In*: *Proceedings of distributed applications and interoperable systems 2003, the 4th IFIP WG6.1 international conference, DAIS 2003, Vol. 2893 of lecture notes in computer science*. Paris: Springer-Verlag, 1–14.
Dey, A.K., 2001. Understanding and using context. *Personal Ubiquitous Computing*, 5 (1), 4–7.
Distributed.net, 2009. *distributed.net: Node Zero* [online]. Distributed Computing Technologies, Inc. Available from: http://www.distributed.net/ [Accessed 17 December 2009].
Dobson, S., *et al*., 2006. A survey of autonomic communications. *ACM Transactions Autonomous Adaptation Systems*, 1 (2), 223–259.
Dowling, J., *et al*., 2005. Using feedback in collaborative reinforcement learning to adaptively optimize MANET routing. *IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans*, 35 (3), 360–372.
E-Ink, 2003. *Electronic paper displays* [online]. Available from: http://www.eink.com/technology/index.html [Accessed 14 July 2010].
Efstratiou, C., *et al*., 2002. Utilising the event calculus for policy driven adaptation on mobile systems. *In*: *3rd international workshop on policies for distributed systems and networks*. Monterey, CA: IEEE Computer Society, 13–24.
Erl, T., 2005. *Service-oriented architecture: concepts, technology, and design,* 1st ed – *cased*. Boston, MA: Prentice Hall.
Fickas, S., Kortuem, G., and Segall, Z., 1997. Software organization for dynamic and adaptable wearable systems. *In*: *ISWC '97: Proceedings of the 1st IEEE international symposium on wearable computers*. Washington, DC: IEEE Computer Society, p. 56.
Flinn, J., *et al*., 2001. Reducing the energy usage of office applications. *In*: *Middleware '01: Proceedings of the IFIP/ACM international conference on distributed systems platforms, Heidelberg*. London, UK: Springer-Verlag, 252–272.
Floch, J., *et al*., 2006. Using architecture models for runtime adaptability. *Software IEEE*, 23 (2), 62–70.
Gabor, K., *et al*., 2003. . An approach to self-adaptive software based on supervisory control. *In*: *Self-adaptive software: applications*, Balatonfured, Hungary: Springer-Verlag, 77–92.
Garlan, D., *et al*., 2002. Project Aura: toward distraction-free pervasive computing. *Pervasive Computing, IEEE*, 1 (2), 22–31.
Garlan, D., *et al*., 2004. Rainbow: architecture-based self-adaptation with reusable infrastructure. *Computer*, 37 (10), 46–54.

Geihs, K., *et al*., 2009a. A comprehensive solution for application-level adaptation. *Software: Practice and Experience*, 39 (4), 385–422.

Geihs, K., *et al*., 2009b. Modeling of context-aware self-adaptive applications in ubiquitous and service-oriented environments. *In*: *Software engineering for self-adaptive systems*. Schloss Dagstuhl, Germany: Springer-Verlag.

Gjørven, E., *et al*., 2006. Self-adaptive systems: a middleware managed approach. *In*: J.P.M.F. Alexander Keller, ed. *2nd IEEE international workshop on self-managed networks, systems & services (SelfMan 2006)*. Dublin, Ireland: Springer.

Grace, P., *et al*., 2008. A reflective framework for fine-grained adaptation of aspect-oriented compositions. *Software Composition*, 4954/2008, 215–230.

Gu, T., Pung, H., and Zhang, D., 2004. Toward an OSGi-based infrastructure for context-aware applications. *Pervasive Computing, IEEE*, 3 (4), 66–74.

Hallsteinsen, S., Floch, J., and Stav, E., 2005. A middleware centric approach to building self-adapting systems. *In*: *Proceedings of software engineering and middleware (SEM 2004), 20–21 September 2004*, 3437/2005. Linz, Austria: Springer-Verlag GmbH.

Hardian, B., Indulska, J., and Henricksen, K., 2006. Balancing autonomy and user control in context-aware systems – a survey. *In*: *PERCOMW'06: Proceedings of the 4th annual IEEE international conference on pervasive computing and communications workshops*. Washington, DC: IEEE Computer Society.

Heiko, A.D., 2007. *Web services differentiation with service level agreements* [online]. IBM. Available from: ftp://ftp.software.ibm.com [Accessed 16 February 2009].

Henricksen, K. and Indulska, J., 2005. Developing context-aware pervasive computing applications: models and approach. *Pervasive and Mobile Computing, In,* 2, 2005.

Henricksen, K., Indulska, J., and Rakotonirainy, A., 2006. Using context and preferences to implement self-adapting pervasive computing applications: experiences with auto-adaptive and reconfigurable systems. *Software – Practice and Experience*, 36 (11–12), 1307–1330.

Hiltunen, M.A. and Schlichting, R.D., 1996. Adaptive distributed and fault-tolerant systems. *International Journal of Computer Systems Science and Engineering*, 11, 125–133.

Hirschfeld, R., Costanza, P., and Nierstrasz, O., 2008. Context-oriented programming. *Journal of Object Technology*, 7 (3), 125–151.

Hoffman, D.M. and Weiss, D.M., eds. 2001. *Software fundamentals: collected papers by David L. Parnas*. Boston, MA: Addison-Wesley Longman Publishing Co., Inc.

Hong, D., *et al*., 2007. Ubiquitous enterprise service adaptations based on contextual user behavior. *Information Systems Frontiers*, 9 (4), 343–358.

IST-MUSIC, 2009. *IST-MUSIC* [online]. IST-MUSIC Consortium. Available from: http://www.ist-music.eu/ [Accessed 4 March 2009].

JBossAOP, 2009. *Framework for organizing cross cutting concerns* [online]. JBossAOP Community team. Available from: http://www.jboss.org/jbossaop [Accessed 13 February 2009].

Kakousis, K., Paspallis, N., and Papadopoulos, G., 2008. Optimizing the utility function-based self-adaptive behavior of context-aware systems using user feedback. *In*: *On the move to meaningful internet systems: OTM 2008*. Monterrey, Mexico: Springer-Verlag, 657–674.

Keays, R. and Rakotonirainy, A., 2003. Context-oriented programming. *In*: *Proceedings of the 3rd ACM international workshop on data engineering for wireless and mobile access*. San Diego, CA: ACM, 9–16.

Keeney, J. and Cahill, V., 2003. Chisel: a policy-driven, context-aware, dynamic adaptation framework. *In*: *POLICY'03: Proceedings of the 4th IEEE international workshop on policies for distributed systems and networks*. Washington, DC: IEEE Computer Society, p. 3.

Kell, S., 2008. A survey of practical software adaptation techniques. *j-jucs*, 14 (13), 2110–2157.

Kephart, J.O. and Das, R., 2007. Achieving self-management via utility functions. *Internet Computing, IEEE*, 11 (1), 40–48.

Kephart, J. and Walsh, W., 2004. An artificial intelligence perspective on autonomic computing policies. *In*: *Policies for distributed systems and networks, 2004. Policy 2004. Proceedings of 5th IEEE international workshop*. Yorktown Heights, NY: IEEE, 3–12.

Ketfi, A., Belkhatir, N., and Cunin, P.Y., 2002. Automatic adaptation of component-based software: issues and experiences. *In*: *PDPTA '02: Proceedings of the international conference on parallel and distributed processing techniques and applications*. Las Vegas, NV: CSREA Press, 1365–1371.

Khan, M., *et al*., 2009. An adaptation reasoning approach for large scale component-based applications. *In*: *Context-aware adaptation mechanism for pervasive and ubiquitous services 2009*, Vol. 19. Electronic Communications of the EASST. Oslo, Norway.

Khan, M.U., Reichle, R., and Geihs, K., 2008. Architectural constraints in the model-driven development of self-adaptive applications. *IEEE Distributed Systems Online*, 9 (7), 1.

Kiczales, G., 1996. Aspect-oriented programming. *ACM Computer Survey*, 28, p. 154.

Kong, J., Jung, J.Y., and Park, J., 2009. Event-driven service coordination for business process integration in ubiquitous enterprises. *Computer Industrial Engineering*, 57 (1), 14–26.

Korpela, E., *et al*., 2001. SETI@home-massively distributed computing for SETI. *Computing in Science and Engineering*, 3 (1), 78–83.

Kortuem, G., *et al*., 2001. When peer-to-peer comes face-to-face: collaborative peer-to-peer computing in mobile ad-hoc networks. *In*: *Peer-to-Peer Computing, 2001. Proceedings of 1st international conference*. Linkoping, Sweden: IEEE, 75–91.

Kurose, J.F. and Ross, K., 2002. *Computer networking: a top-down approach featuring the internet*. Boston, MA: Addison-Wesley Longman Publishing Co., Inc.

Laddaga, R., Robertson, P., and Shrobe, H., 2003. Introduction to self-adaptive software: applications. *In*: *Self-adaptive software: applications*. Balatonfured, Hungary: Springer-Verlag, 275–283.

Leake, D.B., 1996. *Case-based reasoning: experiences, lessons, and future directions*. Menlo Park, CA: AAAI Press/MIT Press.

Leake, D.B. and Sooriamurthi, R., 2002. *Automatically selecting strategies for multi-case-base reasoning*. London, UK: Springer-Verlag.

Ljungstrand, P., 2001. Context awareness and mobile phones. *Personal Ubiquitous Computing*, 5 (1), 58–61.

Long, S., *et al*., 1996. Rapid prototyping of mobile context-aware applications: the Cyberguide case study. *In*: *Proceedings of the 2nd annual international conference on mobile computing and networking*. Rye, NY: ACM, 97–107.

Lutfiyya, H., *et al*., 2001. Issues in managing Soft QoS requirements in distributed systems using a policy-based framework. *In*: *POLICY '01: proceedings of the international workshop on policies for distributed systems and networks*. London, UK: Springer-Verlag, 185–201.

Lymberopoulos, L., Lupu, E., and Sloman, M., 2002. An adaptive policy based management framework for differentiated services networks. *In*: *Policies for Distributed Systems and Networks, 2002. Proceedings of 3rd international workshop*. London: IEEE Computer Society, 147–158.

Maes, P., 1987. Concepts and experiments in computational reflection. *SIGPLAN Not*, 22 (12), 147–155.

Majercik, S.M. and Littman, M.L., 2003. Contingent planning under uncertainty via stochastic satisfiability. *Artificial Intelligence*, 147 (1–2), 119–162.

Manzoni, S., Sartori, F., and Vizzari, G., 2007. Substitutional adaptation in case-based reasoning: a general framework applied to P-truck curing. *Applications of Artificial Intelligence*, 21 (4–5), 427–442.

McKinley, P., *et al*., 2005. *A taxonomy of compositional adaptation*. Technical report, Michigan State University.

Minar, N., *et al*., 2000. Hive: distributed agents for networking things. *Concurrency, IEEE*, 8 (2), 24–33.

Moore, G.E., 2000. Cramming more components onto integrated circuits. *In*: M.D. Hill, N.P. Jouppi, and G.S. Sohi, eds. *Readings in computer architecture*. San Francisco, CA: Morgan Kaufmann Publishers, 56–59.

Noble, B.D. and Satyanarayanan, M., 1999. Experience with adaptive mobile applications in Odyssey. *Mobile Network Applications*, 4 (4), 245–254.

Object Management Group, I., 2009. *OMG, object management group* [online]. Object Management Group, Inc. Available from: http://www.omg.org [Accessed 24 December 2009].

Ohmori, S., Yamao, Y., and Nakajima, N., 2000. The future generations of mobile communications based on broadband access technologies. *Communications Magazine, IEEE*, 38 (12), 134–142.

Pandey, R.K., 2010. Architectural description languages (ADLs) vs. UML: a review. *SIGSOFT Software Engineering Notes*, 35 (3), 1–5.

Papakos, P., *et al*., 2009. VOLARE: adaptive web service discovery middleware for mobile systems. *ECEASST*, 19.

Pascoe, J., 1997. The stick-e note architecture: extending the interface beyond the user. *In*: *Proceedings of the 2nd international conference on intelligent user interfaces*. Orlando, FL: ACM, 261–264.

Paspallis, N., 2009. *Middleware-based development of context-aware applications with reusable components*. Thesis (PhD). University of Cyprus.

Paspallis, N., *et al*., 2009. Developing self-adaptive mobile applications and services with separation-of-concerns. *In*: E.D. Nitto, A. Sassen, and A. Zwegers, eds. *At your service: service-oriented computing from an EU perspective*. Cambridge, MA: MIT Press, 129–158.

Paspallis, N., Kakousis, K., and Papadopoulos, G.A., 2008a. A multi-dimensional model enabling autonomic reasoning for context-aware pervasive applications. *In*: *Workshop for human control of ubiquitous systems (HUCUBIS 2008) in conjunction with the 5th annual international conference on mobile and ubiquitous systems: computing, networking and services (Mobiquitous)*. Dublin, Ireland: ACM Press.

Paspallis, N., *et al*., 2008b. A pluggable and reconfigurable architecture for a context-aware enabling middleware system. *In*: *Proceedings of the 10th international symposium on distributed objects, middleware, and applications (DOA'08)*, Vol. 5331 of *LNCS*. Monterrey, Mexico: Springer Verlag, 553–570.

Passone, S., Chung, P., and Nassehi, V., 2006. Incorporating domain-specific knowledge into a genetic algorithm to implement case-based reasoning adaptation. *Knowledge-Based Systems*, 19 (3), 192–201.

Pawlak, R., *et al*., 2004. JAC: an aspect-based distributed dynamic framework. *Software Practice Experience*, 34 (12), 1119–1148.

Popovici, A., Frei, A., and Alonso, G., 2003. A proactive middleware platform for mobile computing. *In*: *Middleware '03: proceedings of the ACM/IFIP/USENIX 2003 international conference on middleware*. New York, NY: Springer-Verlag, 455–473.

Popovici, A., Gross, T., and Alonso, G., 2002. Dynamic weaving for aspect-oriented programming. *In*: *AOSD '02: Proceedings of the 1st international conference on Aspect-oriented software development*. New York, NY: ACM, 141–147.

Raverdy, P.G. and Lea, R., 1999. Reflection support for adaptive distributed applications. *In*: *Enterprise distributed object computing conference, 1999. EDOC '99. Proceedings. 3rd international*. IEEE, 28–36.

Romero, D., *et al*., 2009. Enabling context-aware web services: a middleware approach for ubiquitous environments. *In*: Y. Michael Sheng, Jian, and D. Schahram, eds. *Enabling context-aware web services: methods, architectures, and technologies*. Lille, France: Chapman and Hall/CRC, 113–135.

Rouvoy, R., Beauvois, M., and Eliassen, F., 2008a. Dynamic aspect weaving using a planning-based adaptation middleware. *In*: *MAI '08: Proceedings of the 2nd workshop on Middleware-application interaction*. New York, NY: ACM, 31–36.

Rouvoy, R., Conan, D., and Seinturier, L., 2008b. Software architecture patterns for a context-processing middleware framework. *IEEE Distributed Systems Online*, 9 (6), 1.

Rouvoy, R., *et al*., 2009. MUSIC: middleware support for self-adaptation in ubiquitous and service-oriented environments. *In*: B.H.C. Cheng, R. Lemos, H. Giese, P. Inverardi, and J. Magee, eds. *Software engineering for self-adaptive systems*, Vol. 5525 of *Lecture Notes in Computer Science*. Berlin, Heidelberg: Springer Berlin Heidelberg, chap. 9, 164–182.

Rubin, A., 2008. *The future of mobile* [online]. Available from: http://googleblog.blogspot.com/2008/09 /future-of-mobile.html.

Ruiz, P., Botia, J., and Gomez-Skarmeta, A., 2004. Providing QoS through machine-learning-driven adaptive multimedia applications. *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions*, 34 (3), 1398–1411.

Russell, S.J. and Norvig, P., 2002. *Artificial intelligence: a modern approach* (International Edition). Pearson US Imports & PHIPEs.

Salber, D., Dey, A.K., and Abowd, G.D., 1999. The context toolkit: aiding the development of context-enabled applications. *In*: *CHI '99: Proceedings of the SIGCHI conference on human factors in computing systems*. New York, NY: ACM, 434–441.

Satyanarayanan, M., 2001. Pervasive computing: vision and challenges. *Personal Communications, IEEE*, 8 (4), 10–17.

Schilit, B., Adams, N., and Want, R., 1994. Context-aware computing applications. *In*: *Workshop on Mobile Computing Systems and Applications, 1994. Proceedings*, IEEE, 85–90.

Schmidt, A., 2000. Implicit human computer interaction through context. *Personal and Ubiquitous Computing*, 4 (2), 191–199.

Scholz, U. and Rouvoy, R., 2008. Divide and conquer – organizing component-based adaptation in distributed environments. *ECEASST*, Article 5, 11.

Shanahan, M., 1999. The event calculus explained. *Artificial Intelligence Today: Recent Trends and Developments*, 409–430.

Sindico, A. and Grassi, V., 2009. Model driven development of context aware software systems. *In*: *International workshop on context-oriented programming*. Genova, Italy: ACM, 1–5.

Smith, D.E. and Weld, D.S., 1999. Temporal planning with mutual exclusion reasoning. *In*: *IJCAI '99: Proceedings of the 16th international joint conference on artificial intelligence*. San Francisco, CA: Morgan Kaufmann Publishers Inc, 326–337.

Sousa, J.P. and Garlan, D., 2002. Aura: an architectural framework for user mobility in ubiquitous computing environments. *In*: *WICSA 3: Proceedings of the IFIP 17th world computer congress – TC2 stream/3rd IEEE/IFIP conference on software architecture*. Deventer, The Netherlands: Kluwer, B.V., 29–43.

Sousa, J., *et al*., 2006. Task-based adaptation for ubiquitous computing. *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, 36 (3), 328–340.

Strang, T. and Linhoff-Popien, C., 2004. A context modeling survey. *In*: *Workshop on advanced context modelling, reasoning and management, UbiComp 2004 – the sixth international conference on ubiquitous computing*. Nottingham, UK: Citeseer.

Strassner, M. and Schoch, T., 2003. Today's impact of ubiquitous computing on business processes. *In*: *First international conference on pervasive computing*, Springer, 62–74.

Sutton, R.S. and Barto, A.G., 1998. *Reinforcement learning: an introduction (adaptive computation and machine learning series)*. Cambridge, MA: The MIT Press.

Szyperski, C., 1997. *Component software: beyond object-oriented programming*. Addison-Wesley Professional.

Tesauro, G. and Kephart, J.O., 2004. Utility functions in autonomic systems. *In*: *Proceedings of the 1st international conference on autonomic computing*, New York: IEEE Computer Society, 70–77.

van der Heijden, H., 2003. Ubiquitous computing, user control, and user performance: conceptual model and preliminary experimental design. *In*: U. Lechner, ed. *Proceedings of the research symposium on emerging electronic markets*. Bremen, Germany: University of Bremen, 107–112.

Wang, P., 1996. The interpretation of fuzziness. *IEEE Transactions on Systems, Man, and Cybernetics*, 26, 312–326.

Want, R., *et al*., 1992. The active badge location system. *ACM Transactions on Information Systems*, 10 (1), 91–102.

Weiser, M., 1993. Ubiquitous computing. *Computer*, 26 (10), 71–72.

Weiser, M., 1999. The computer for the 21st century. *SIGMOBILE Mob. Computing Communications Review*, 3 (3), 3–11.

Weiser, M. and Brown, J.S., 1997. The coming age of calm technolgy. *In*: P.J. Denning and R.M. Metcalfe, eds. *Beyond calculation: the next fifty years*. New York: Copernicus, 75–85.

Wright, A., 2009. Get smart. *Communications of the ACM*, 52 (1), 15–16.

Zachariadis, S., 2005. *Adapting mobile systems using logical mobility primitives*. Thesis (PhD). Department of Computer Science, University College London.

Zachariadis, S. and Mascolo, C., 2006. The SATIN component system – a metamodel for engineering adaptable mobile systems. *IEEE Transactions on Software Engineering*, 32 (11), 910–927.

Zachariadis, S., Mascolo, C., and Emmerich, W., 2003. Adaptable mobile applications: exploiting logical mobility in mobile computing. *In*: *Mobile agents for telecommunication applications*. Heidelberg: Springer Berlin, 170–179.