ACTUS: a Framework for Adaptation Control in Ubiquitous Computing

Luciano Cavalheiro da Silva*, Cristiano Costa, Cláudio F. R. Geyer
Instituto de Informática – UFRGS – Porto Alegre – RS – Brasil
{lucc, cacosta, geyer}@inf.ufrgs.br

Abstract

The moving of Human-Computer Interaction towards "one user, many devices", combined to the raising of mobility-related technologies, inspired a vision of the computing becoming invisible to its users by its seamless integration into the users day-life tasks – the birth of Ubiquitous Computing (UbiComp) research. A core subclass of UbiComp applications is that of context-aware applications. Such applications adapt their behavior to the prevailing (dynamic) resource availability levels in the environment, aiming to optimize user-environment interactions and to reduce the demand for user intervention. The problem of adaptation controlling is related to orchestrating adaptations carried out by concurrent applications, in order the promote system stability, while considering other desirable properties as agility and relative priorities. In this paper we present ACTUS, a proposal of framework for building adaptation controllers targeted at UbiComp.

1. Motivation

Nowadays, we observe the smooth merging of mobile computing technologies into our day-life activities. Related to this raising of mobility technologies, a research area of increasing interest is *ubiquitous computing*, often referred also as *pervasive computing* [13]. Term coined originally by Mark Weiser in the late 1980's [16], *ubiquitous computing* (*UbiComp*) describes a proposition grounded on the vision of turning the computing invisible by its seamless integration into the user's day-life tasks.

Despite its rather idealized meaning, the property of *invisibility* has many manifestations in real-world ubiquitous computing research. It is, perhaps, best understood as a *distraction-free* pervasive environment, emphasizing that the user attention becames the most valuable resource in the system. [7]. Another interpretation, highlighting the mobility aspect, would be a *follow-me semantics* for applications and data [2], enabling a continuated execution of the user's tasks with access from any-where, at any-time and using whatever device is currently available, despite of the user (physical) mobility. Either way, common assumptions are (i) that adaptation is a mandatory feature and (ii) that the user must be shielded from environment management and adaptation control concerns, as much as possible, in order to preserve the so desired invisibility.

This paper presents ACTUS, a proposal of framework for building adaptation controllers targeted at Ubiquitous Computing. The remaining of this paper is organized as follows. Section 2 discusses the design of adaptive ubiquitous applications, highlighting the issues related to the adaptation control problem, and briefly revisits the state of art. Following, section 3 introduces the ACTUS framework, outlining its architecture, as well as discusses aspects of its data models for context and adaptation and their handling of application evolution. Finally, section 4 presents some concluding remarks.

2. Context-aware Ubiquitous Applications

The availability of ubiquitous network access, yet not continuous in time, shall modify the profile of running (end-user) applications, which shall evolve from transient activations into virtually continuous executions where the application is aware of its current context and modifies aspects of its execution with the aim of optimizing the user interactions with the environment. Such changes in the application's context may be caused by physical displacements of the user, by modification of the user's currently active activities, as well as in function of the resource availability changes in system, which is also dynamic[14].

Hence, an important subclass of ubiquitous computing applications, and the object of our research, is that of smart, adaptive (context-aware) ubiquitous applications. In fact, we believe that, concerning to goal of invisibility, context-aware adaptation is a mandatory feature. However, what does context exactly means? A widen-accepted definition for "context" is given by [6], who defines it as "any information that can be used to characterize the situation of an entity", whereas entity may be "a person, place, or object that is considered relevant to the interaction between a

* This author is supported by CNPq, Brazil, through the PhD student grant number 142391/2005-0.

user and an application, including the user and the application themselves". Notice that, the notion of context includes information about the state of the hardware and software infrastructures, as well as about the user activities and intentions that might be relevant to the execution of a given application being studied.

2.1. Design Challenges for Context-awareness

The task of building context-aware applications may decomposed into three perspectives:

1. *programming of adaptive behaviors* – concerned about which models and abstractions one must use to embed adaptive behaviors into context-aware applications;
2. *context engineering* – concerned about how to model context (at design time) and how to acquire, process and deliver the relevant information (at run time); and
3. *adaptation control* – given that we are able to realize changes in the context and that we are aware of the existence of alternative adaptation behaviors, this perspective is concerned on deciding which adaptation to activate next.

The first two perspectives are easier to be identified as problems because they have direct impact on every context-aware application, not only those related to ubiquitous computing. However, the third one, which corresponds to the *problem of adaptation control in ubiquitous computing*, increases in importance as we start looking to the system as a whole and considering the intrinsic characteristics of the ubiquitous computing environment where those applications execute, as scale and resource sharing.

Increasing in scale is a legacy of Weiser's prospect on "computing invisibility" [16]: to be invisible the (ubiquitous) computing environment should not limit users mobility, so it grows spreading over the physical ambient by incorporating new devices and services, such that the computing environment is always available, up to some degree, to the user. In turn, large scale computing environments tend to be shared infrastructures to be economically viable. As a side effect, one would expect the number of applications executing in this environment to be large and the system to be very dynamic.

Moreover, another implication of being a shared computing environment is that adaptations taken in one application affect the other applications executing in the system. Thus, the improper mediation of such adaptations, which is raised by the scale of the system, may lead, at least, to the wasting system resources and, in the worst case, to such a level of system instability that compromises invisibility.

2.2. State of Art Revisited

The first two design views outlined in section 2.1 have already been approached by quite some works so far. Related to programming, we may cite the works of [1, 8, 9], whereas related to context engineering, task which is typically accomplished through middleware, we may cite [3, 4, 15, 5]. On the other hand, despite the seminal work of Noble[10, 11], the problem of adaptation control in shared environments remains an open question for research. The solution proposed by Noble, thought addressing the multi-application shared environment scenario, was specific to data adaptation.

It is worth to note that, yet not explicitly addressed, the problem of adaptation control still exists in the other works, but the approach falls short with respect to scalability and flexibility. There the control is typically hard-coded into the application or implicitly defined in context recognition process. Moreover, the adaptation policy is "always execute the adaptation", which is not satisfactory in large scale shared environments since is clearly neglects the side-effects between applications and the different demands of priority of adaptations, as well as do not account for the case of multiple alternative adaptations being available for a given context change.

3. The ACTUS Framework

Aiming to address the problem of adaptation control in ubiquitous computing outlined in section 2.1 we propose ACTUS which is a framework for building adaptation controllers.

By designing ACTUS as a framework, we aim to provide a generic skeleton solution which may be customized for specific adaptation control demands (e.g., different heuristics for adaptation scheduling). Further, ACTUS is meant to be language-neutral and application-neutral, therefore supporting a wide-range of application types and adaptation patterns. Besides, ACTUS is not concerned about context recognition. Instead, it is meant to work in a complementary fashion to third-part solutions for acquiring context information, cooperating with them to support adaptation behaviors at the applications. Another important concern in the design of ACTUS was to account for application evolution in long term ubiquitous applications.

Reaching the desired level o generality for the framework goes through defining suitable models for representing context sensibility and adaptation features of the applications. Specifically, those models should expose to the adaptation controller information that is pertinent to the adaptation control decision process while hiding other implementation specific details (e.g. programming language, context-recognition middleware). Those requirements are addressed by the ACTUS context and adaptation models.

3.1. Architecture of ACTUS

As the figure 1 shows, from a macro perspective, ACTUS is composed by two main components: the *adaptation con-*

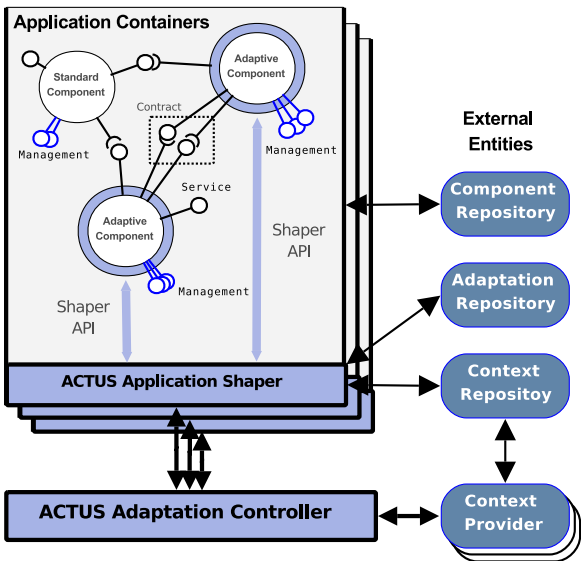

Figure 1. Macro-architecture of ACTUS

troller itself and many instances of the *application shaper*. Besides, ACTUS assumes the existence of some external entities, which complement its functionality in order to fully support for adaptive executions. Among those external entities, we highlight the repositories for adaptation actions and context information, as well as the context providers. The code repository, though not strictly necessary for the framework, evoques the assumption of evolution of the applications, i.e. their hability to incorporate new features over time during their execution.

The *adaptation controller*, which is shared between applications in order to enable orchestration of adaptations, is responsible by the decision making process which results in the scheduling of adaptation actions. In this process, it uses information exposed by the application shapers, as well as, events describing changes in the observed context state, which are issued by registered context providers.

The *application shaper*, on the other hand, exists in a per-application basis and has a dual nature. From the application point of view, the application shaper represents a language-specific binding of the ACTUS API. The application uses such API in order to expose its adaptation features to the adaptation controller without being concerned on how to reach the adaptation controller nor on data representation issues. The shaper utilizes the *context* and *adaptation models* defined in ACTUS in order to represent the sensitiveness and the adaptation capabilities of the application in a standard way, suitable to be used by the adaptation controller. In turn, from the controller point of view, the shaper is responsible by forwarding the adaptation hints issued by the adaptation controller to the corresponding internal elements of the application, shielding the controller from the complexity of dealing with implementation specific issues of the application.

3.2. Context Model

The *context model* represents the application's perceptions about its execution environment. It includes all the relevant information that the application is able to use in order to adapt its execution to the prevailing environmental conditions. In our formal definition, the context model of an application is represented by a set of dynamically evaluated variables, which we call *context elements*.

Once the environment state described by the context model changes *significantly*, the application modifies aspects of its execution accordingly, i.e. it adapts in order to optimize future interactions. Yet, it is worth to notice that such changes may be either reflex of real observations or forecasted conditions, meaning that adaptation may occur both reactively and proactively. The specific way the application reacts to changes in the state of its context model is defined in its *adaptation model.*

Definition. We define κ_{basic}, our *basic context model* for an Ubiquitous Computing application (or application component), as a dual representation of low-level and high-level contextual information,

$$\kappa_{basic} = (E, C)$$

, with $E = \{e_0, e_1, \ldots, e_m\}$, and $C = \{c_0, c_1, \ldots, c_n\}$. The former tuple element, E, characterizes the (generic) low-level state information available to the application through a set of uniquely identified raw sensors, e_i, provided in the environment. Those environment sensors may be both hardware and software sensors, and are typically shared across several applications. The later tuple element, C, corresponds to the high-level (specialized) contextual information, given as a set of context element c_k, that the application effectively uses to decide about adaptation processes. In turn, we define the context element c_k as a tuple

$$c_k = (k, S_k, \sigma_k(\phi))$$

, where:

- k is the unique identifier of context element c_k in the system;
- $S_k = \{s_0, s_1, \ldots, s_n\}$ is the finite set of context states of c_k;
- $\sigma_k : \phi \rightarrow S_k^*$ is a function which actually employs the context recognition process for context element c_k, reading relevant input sensors, $\phi \subseteq E$, and translating the obtained data into abstract context element states.

The κ_{basic} definition represents a superset of the studied approaches for context modeling in ubiquitous computing systems that we have found in the literature. Specifically, to be flexible and general with respect to the context recognition process, and also as a preliminary accounting for uncertainty, we assume the output of $\sigma_k(\phi)$ to be a tuple of *enabled* context states instead of a single context state It is left up to the adaptation controller to later decide which context state to use, considering the adaptation model of the application.

We should highlight that both E and $\sigma_k(\phi)$ are, in fact, opaque handles for external entities which are provided by tools that are outside the framework. Hence, from the framework point of view, it matters neither how the sensors actually gather their corresponding information nor how the context recognition process actually is carried out, given that they provide the expected semantics. Those entities are, however, represented in the framework because they impact on the context model evolution process, as we describe later.

3.3. Evolution in the Basic Context Model

Informally, the evolution of the context model may be caused either by a modification in the sensors available in the environment or by a change in the context elements that affect the application execution. In the later case, such a change would either modify the possible states of a given context element or modify the strategy used to infer such states from the raw level information gathered from the environment sensors.

To reduce the complexity of the decision process accomplished by the adaptation controller, we put a constraint of *monotonicity* [12, p. 212] in the model evolution process, i.e., the context model can only *increase* in information. This way, we avoid the complexity of invalidation procedures at the adaptation controller required to keep the consistency of the internal data structures in case that some definition in the model would have disappeared or changed its meaning. Thought it would appear restrictive at a first glance, since in the real world application context things may cease to exist, we argue it is not the case, because the *unknown* value may be used in place of information that is no longer available in the system. This assumption leads to a cleaner model, since the *unknown* value should already be handled at the context recognition procedure in order to account for uncertainty.

Definition. Be K the set of all context models κ. We then define the *context model evolution function*

$$Evolve_\kappa(\kappa, \delta_\kappa) : K \times K \rightarrow K$$

,which evolves the definition of the context model κ into a new one, by applying a differential context model update δ_κ. Notice that, given the monotonical evolution constraint, δ_κ is itself a valid context model, as it includes both context element and environment sensor definitions. Thus, the evolution process is, in essence, a recursive merging of both context models definitions, as algorithm 1 shows.

Algorithm 1 Abstract defintion of $Evolve_\kappa(\kappa, \delta_\kappa)$

```
1  function Evolve_κ(κ, δ_κ):κ
2    E, C ← κ
3    E', C' ← δ_κ
4    E ← E ∪ E'    // import new sensors
5    for c_k' ∈ C' do   // merge context elements
6      if ∃c_k ∈ C | k = k' then  // update existing
7        k', S', σ' ← c_k'
8        k, S, σ ← c_k
9        S ← S ∪ S'    // merge state sets
10       if σ' ≠ ∅ then
11         σ ← σ'  // replace context source
12       end if
13     else      //   new context element
14       C ← C ∪ {c_k'}
15     end if
16   end for
17   return κ
18 end function
```

3.4. Adaptation Model

In general lines, the purpose of the *adaptation model* is to provide directions and tools that enable the adaptation controller to keep the application in an *adapted operation mode* as much time as possible.

It comprises an *adaptation policy* and some *adaptation actions*. The former establishes criteria for evaluating whether the application is well adapted to a given observed execution context. In turn, the later defines the capabilities of further adaptation that exist in the application and it is specified in the form of adaptation behaviors (actions) bound to some subset of the defined states of relevant context elements. In fact, adaptation behaviors may also be bound to specific transitions between context states for a finer tuning of adaptation (e.g., avoid some unnecessary reconfigurations when the original and the target states share some operation aspects). Figure 2 shows an example context model anotated with adaptation triggers.

Before we proceed with detailing of the adaptation model, it is worth to get a better understanding on the meaning of an application operation mode.

Definition. Be $\kappa = (E, C)$, the context model for an ubiquitous application (as defined previously in section 3.2). We, then, define

$$O = \{(c_o, s_a), (c_1, s_b), ..., (c_n, s_m)\}, \forall c_i \in C$$

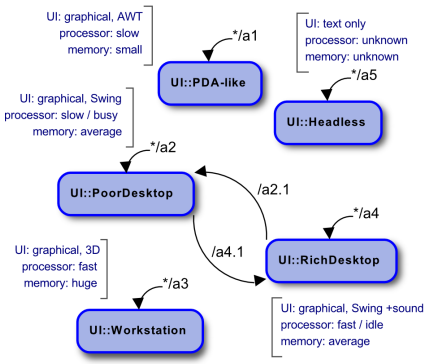

Figure 2. Annotated Context Model

as the *operation mode* for such ubiquitous application, from the adaptation control point of view. It is described by bindings of specific states for every context element defined in the context model of the application. Therefore, to be strictly consistent, we should add constraints in the form $s_a \in States(c_o)$, $s_b \in States(c_1)$, ..., $s_n \in States(c_m)$.

Formerly, a state bound to a given context element represents the last *kind of adaptation* implemented with respect to changes in that context element state. Based on this notion, following we introduce the property of consistency for operation modes.

Definition. We say the operation mode is *consistent* with respect to some observed environment if, for every context-element state binding in the that operation mode, the bound state is also observed in that environment.

A consistent operation mode means that, with that configuration (result of executing some adaptation actions), the application is able to properly execute under that observed environmental conditions. It is also worth to notice that, for a given application, there would be many consistent operation modes for a given observed environment state and that not all consistent operation mode will be optimal with respect to resource utilization and user productivity.

Definition. We define μ_{basic} our *basic adaptation model* for an ubiquitous computing application as a triplet

$$\mu_{basic} = (\kappa, A, \varphi_{basic}(E, c, o))$$

, where

- $\kappa = (E, C)$ is the related context model;
- $A = \{a_0, a_1, ..., a_m\}$, is a set of uniquely identified adaptation actions $a_i = (b_i, T)$, each of them describing a distinct adaptation behavior (b_i) supported at the application; $T = \{t_0, t_1, ..., t_n\}$, is a set of triggers $t_j = (c_k, s_a, s_b) | \, c_k \in C \wedge \; s_a, s_b \in S(c_k)$, which describes the circumstances (context state transitions) in which action a_i would be executed to improve some aspect of the application execution;
- $\varphi_{basic}(E, c, o) : E \times C \times S(c) \rightarrow S(c)^*$, is a *stateless function* which implements the *adaptation policy* of the application, suggesting alternative operation modes for the adaptation controller, considering the currently observed environmental conditions.

Adaptation actions encapsulate features implemented outside the adaptation controller, and for which complete and correct information is not available in the system. We assume, however, that it is possible to differentiate between actions because they have a characterizing property, their behavior, which is unique among all other actions defined in the system. Thought two behaviors may produce the same effects under determined conditions, we are not concerned about determining action equivalence. That is, if two actions share exactly the same behavior (implementation), they are the same action from the controller point of view, otherwise they are taken as distinct actions, no matter their final effects are, apparently, always the same.

The *basic adaptation policy* φ_{basic}, while provider of directions for the operation of the adaptation controller, is a policy that would ensure that the application keeps executing despite changes in the environment, but which is not concerned about fine-tuning the application behavior in order to optimize the user productivity. This idea may be formulated in terms of the concept of consistent operation mode as "keep the application at any consistent operation mode".

3.5. Evolution in the Basic Adaptation Model

Regarding to the formulation for the basic adaptation model provided in section 3.4, we may see evolution of the adaptation model happening in three ways: (i) evolution of the related context model; (ii) evolution of the set of adaptation actions A ; or (iii) evolution of adaptation policy φ. Similarly to the case of context model evolution, we would like to put a constraint of monotonicity in the adaptation model evolution process, aiming to reduce the complexity of the logic implemented at the adaptation controller. Therefore, the adaptation model, likewise the context model, should only increase in information.

The first case, the evolution of the context model, is then straightforward to handle: we just apply the function $Evolve_\kappa$ previously defined. Concerning to the set of adaptation actions A and the constraint of monotonicity, it is also straightforward to realize that we may allow the definition of new adaptation actions, which are included in the set A, since it only increases the amount of information in the model. Conversely, we must deny any deletion of actions, since it would cause the amount of information to shrink. Nevertheless, issues arise when we decide about which kind of modifications we should allow for already defined adaptation actions.

Algorithm 2 Abstract definition of $Evolve_{\mu}(\mu, \delta_{\mu})$

```
1  procedure Evolve_μ(μ, δ_μ)
2    κ, A, φ ← μ
3    δ_κ, A', φ' ← δ_μ
4    Evolve_κ(κ, δ_κ)   // evolve the context model
5    for ∀a'_j ∈ A' do
6      if ∃a_i ∈ A | i = j then
7        T(a_i) ← T(a'_j)   // replace triggers
8      else
9        A ← A ∪ {a_j}
10     end if
11   end for
12   if φ' ≠ ∅ then   // update the adaptation policy
13     φ ← φ'
14   end if
15 end
```

Since the behavior is the characterizing property (the identity) of an adaptation action, we take it as an invariant property of that action. However, we understand there will be situations (e.g., fixing bugs), the application developer will like to completely replace an existing action with another one and ensure the older action to never be executed again. Thought one can't directly modify the behavior of an action, it is possible to define a new action corresponding to the modified behavior. It solves half of the problem. Nevertheless, the issue of avoiding further executions of the action remains. It may be tackled through three schemes:

- *pre-processing* – it consists in suppressing those inputs, formerly context-state transition events, that would cause the activation of undesired actions;
- *in-processing* – it consists in modifying the controller logic to avoid it outputting undesired adaptation actions; or
- *post-processing* – it consists in filtering-out undesired adaptation actions output by the controlled before they are effectively put into execution.

After evaluating the benefits and drawbacks of each approach, which we omit in this paper due to space limitations, we have selected *pre-processing* of of inputs, since it leads to a the least resource comsuption at the controller (mainly, by avoiding it to reason about actions the will never be executed), what is an important property concerning to the scalability of the solution. Specifically, our approach is to enable the deletion of triggers from action definitions.

4. Concluding Remarks

In this paper we have presented ACTUS, our proposition of generic framework for build adaptation controllers target to ubiquitous computing systems. We have outlined its macro architecture and have focused on aspects of its data models for context and adaptation, and their mechanisms to accommodate application evolution. Due to space limitations, however, other aspects of the framework would not be covered. Among them we cite the control model and the extensions for the data models which would enable quality gain at the decision making carried by the adaptation controller. Those topics, as well as experimental results gathered from a prototype which is under development, will be approached in another paper.

References

[1] I. Augustin. *Abstrações para um Linguagem de Programação Visando Aplicações Móveis em um Ambiente de Pervasive Computing*. Doutorado em ciência da computação, Instituto de Informática, CPGC/UFRGS, Porto Alegre, Janeiro 2004.

[2] I. Augustin, A. C. Yamin, J. L. V. Barbosa, L. C. da Silva, R. A. Real, G. Frainer, G. G. H. Cavalheiro, and C. F. R. Geyer. *Mobile Computing Handbook*, chapter ISAM, Joining Context-awareness and Mobility to Building Pervasive Applications, pages 73–94. CRC Press, New York, 2004.

[3] G. Chen. Solar: Building a context fusion network for pervasive computing, 2004.

[4] R. C. A. da Rocha and M. Endler. Middleware: Context management in heterogeneous, evolving, ubiquitous environments, 2006.

[5] P.-C. David and T. Ledoux. Wildcat: a generic framework for context-aware applications. In *MPAC*. ACM, 2005.

[6] A. K. Dey. Understanding and using context. *Personal and Ubiquitous Computing*, 5(1):4–7, 2001.

[7] D. Garlan, D. Siewiorek, A. Smailagic, and P. Steenkiste. Project aura: Toward distraction-free pervasive computing. *IEEE Pervasive Computing*, 01(2):22–31, 2002.

[8] K. Henricksen and J. Indulska. Developing context-aware pervasive computing applications: Models and approach, 2005.

[9] J. Munnelly and et al. An aspect-oriented approach to the modularization of context, 2007.

[10] B. Noble. System support for mobile adaptive applications. *IEEE Personal Communications*, 7(1):44–49, fev 2000.

[11] B. Noble, M. Satyanarayanan, D. Narayanan, J. E. Tilton, J. Flinn, and K. R. Walker. Agile application-aware adaptation for mobility. In *16th ACM Symposium on Operating System Principles (SOSP 97)*, pages 276–287, 1997.

[12] S. Russel and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2003.

[13] D. Saha and A. Mukherjee. Pervasive computing: a paradigm for the 21st. century. *IEEE Computer*, 36(3):25–31, mar 2003.

[14] M. Satyanarayanan. From the editor in chief: The many faces of adaptation. *IEEE Pervasive Computing*, 03(3):4–5, 2004.

[15] K. Sheikh and et al. Middleware support for quality of context in pervasive computing, 2007.

[16] M. Weiser. The computer of the 21st century. *Scientific American*, 265(9), Sept. 1991.