Engineering Adaptation: A Component-based Model

Nikola Šerbedžija
Fraunhofer FIRST
Berlin, Germany
nikola.serbedzija@first.fraunhofer.de

Abstract—**The novel concept of user-centric pervasive adaptive systems has been designed to deliver services adapted to our needs and wishes according to the context of use. Engineering adaptation is a cross disciplinary endeavour requiring synergy of computer and human sciences as well as the practice. This work describes a novel *reflective approach* for development and deployment of pervasive adaptive systems. Special focus is on *reflective architecture* which uses component and service based programming model for developing the reflective framework as a generic support for the pervasive adaptive systems. A strong pragmatic orientation of the component-based approach is illustrated by a prototype named affective music player.**

Keywords–Adaptive Systems; Autonomous Behaviour; Component-based Systems.

I. Introduction

Seamless and implicit human-computer interaction is an important characteristic of smart technology [1]. The 'smart' attribute is achieved by intuitive system control, based on the context assessment. This allows the system to function autonomously, without the requirement for explicit user intervention. In other words, the user becomes a part of the control loop, making system reaction adaptive and appropriate to users' behaviour.

Most of the present systems that deal with personal human experience are poorly engineered [1]. As a consequence, maintenance and modification of smart applications are very difficult. Furthermore, re-usability as capability of re-deploying the same software structures in different application domains is not possible. The presented reflective approach adds complexity to the current pervasive adaptive[2] systems by introducing seamless and implicit man-machine interaction based on emotional, cognitive and physical experience. At the same time it strives at generic, flexible and re-usable solutions that should overcome the poor engineering problems.

In effort to mimic the adaptation process, as it appears in the nature, and to apply it within man-machine interaction, reflective approach deploys the biocybernetic loop to make users' psychophysiological data a part of computer control logic [2][3]. The function of the loop is to monitor changes in user's state in order to initiate an appropriate computer response. This approach also takes results of affective/physiological computing [3] and combines it with high level understanding of social and goal–oriented situations. Biocybernetic loop [4] is implemented with the help of sense-analyze-react control troika. Firstly, reflective ontology [5] classifies numerous factors that determine user's states, social situation and application goals, defining elements for decision making. The ontology is then expressed in a number of XML-based taxonomies that allow for a uniform deployment in data acquisition, user's state diagnoses and activation of corrective actions. Finally the component based framework is developed with a goal to support adaptation process and deployment of adaptive applications [6].

The rest of the paper focuses on software engineering strategies of reflective approach. Firstly, the adaptation concept is presented, followed by technical blueprints of the component based architecture for adaptive man-machine interaction. Finally, the application of this approach is illustrated by the prototype music player, a system that controls the player according to the listener emotional state. The conclusion summarizes the work described and indicates further challenges and research topics in the domain of adaptive systems.

II. Adaptation Strategy

The overall goal of reflective systems [7] is to create a software framework that controls and adapts the environment (a home, an office or an automotive environment) according to the users' situation. To be able to perform this task, a system must be able to perceive its environment through sensors and influence it through actuators. Therefore, a reflective application always consists of hardware (sensors and actuators) and software (reading sensor values, controlling the application and operating actuators) that together with users build the application context.

Controlling the environment through software can be done in two different ways: as feed-forward system (Figure 1a, also called open control loop), or as feedback system (Figure 1b, also called closed control loop).

A feed-forward system does not take into account the reactions of the system under control, but only the environment under which it operates: in the reflective domain, this would amount to the control of the environment without observing the reactions of the user.

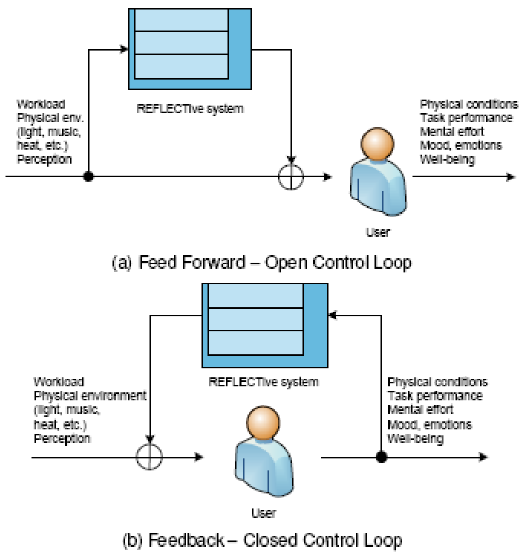


Figure 1. User centric system

Feed-forward systems offer quick response and high performance with few needed sensors – if they can be implemented. The main difficulty in creating feed-forward systems is that the impact and effects of the environment and the controlled actuators must be precisely known, as they are not measured. E,g. the effects of changing the lighting in a room must be completely known under all conditions – obviously, a task too challenging and far-fetched for now. Instead, this approach focuses on creating feedback control systems, measuring and reasoning on its effects and the current behaviour of the user. By doing this, a reflective system may also become able to counter the effects of unknown factors influencing the users' well-being.

Feed-forward systems are sufficient for a user-friendly behaviour in situation where satisfying an average user goals are needed (e.g., word processing systems, etc.). However, as reflective systems should be sensitive to a personal user state (emotional, cognitive and physical) policy of "satisfying an average user" never works, as each user is different and the behaviour of the single user often differs in different circumstances. If a system is to meet more personal user needs, a self-tuning and self-correcting strategy is a *conditio sine qua non.*

Even if building of closed control loops instead of open control loops seems to be a more promising approach, it brings along additional challenges. First of all, in the setting that is investigated, it is never guaranteed that an action performed on the environment will show the desired effects or any effects at all: preferences vary greatly over persons and over time. The reactions to hard rock and rap music differ between individuals, but also the reaction to a certain lighting condition may differ depending on the user's context. Reflective systems will therefore have to self-assess their effects on their environment, and hold several alternative strategies ready to achieve their goal. Furthermore, extracting the user's conditions from low-level psycho-physiological measures or computing it through image processing is a challenging task with varying success depending on the user's context; detecting the facial expression under bad lighting conditions for example is very difficult, if not impossible.

These challenges call for a well-structured software solution that allows for flexible response to the user's environment, and self-assessment of its performance. In any case (both feed-forward and feed-back control), a troika "sense-analyse-react" needs a special consideration. The closed loop control for implicit human-machine interaction, based on user psycho-physiological state is often denoted as a biocybernetic loop [2][4]. The function of the loop is to monitor changes in user state in order to initiate an appropriate adaptive response. The biocybernetic loop is designed according to a specific rationale, which serves a number of specific meta-goals. For instance, the biocybernetic loop may be designed to: (1) promote and sustain a state of positive engagement with the software/task and (2) minimise health or safety risks inherent within the human-computer interaction.

Both biocybernetic loop deployment and reflective ontology that supports high level reasoning have been described elsewhere [4][5], as well as some of the reflective applications which deploy this technology [6][7]. This paper focuses on reflect component model and its organization.

III. Reflective Component Architecture

The programming paradigm for building reflective applications is component-based [8]. Software components are units of software that make their communication capabilities and requirements explicit by means of ports: provided ports describe what communication the component can accept and process, while required ports describe the communication the component requires to perform its work. Ports are given types that describe the set of messages that can be received or sent. A component based system is comprised of a set of components and an assembly that describes the way they are connected; required ports can be connected to the provided ports with the same type by a connector.

Having components that make all communication requirements and capabilities explicit helps in several ways: this provides a simple framework for reusability of components. At the same time, components can be written without considering one specific application. A component is written with just its own communication in mind, and, at a later point and possibly by a different person, it is made part of an application. Hence, components are usually more generic than special-purpose algorithms and can be employed in different applications. Furthermore, a component-based system can be reconfigured at run-time [9].

This reconfiguration can be parametric, i.e., the reconfiguration is performed by changing parameters of components that affect their behaviour. Reconfiguration can be also structural, i.e. the change of behaviour is achieved by removing components, adding components, or changing the interconnections between components.

Both kinds of reconfiguration allow to realize short-, mid-, and long-term adaptation of the system – as needed for different biocybernetic loops (as described elsewhere) [4][5]. Some components do not require communication partners, but merely provide communication. This especially holds true for wrappers of hardware devices like sensors or actuators. In accordance with common terminology, such components are called services.

Since services do not require communication partners, suitable services can be chosen by considering only the ports they provide. Especially for sensors and actuators, this enables automated discovery as well as dynamic response to the availability of new services on behalf of the architecture.

A. *Reflective Framework*

Reflective systems are structured as feedback loops that sense the environment, analyse the gathered data, and react according to the analysed results. Consequently, applications are structured in three layers:

- The tangible layer,
- The reflective layer, and
- The application layer.

The detailed description of the reflective layered architecture can be found in [7][10]. The focus here is on reflective layer and its component-based organization. It forms a pool of reusable software components that can be used to analyse the set of features exposed by sensors, and components to coordinate actuators well as a generic reasoner and learning algorithm components. On the one hand, reasoners are used to provide a mapping from the current user and environmental state to plans. Learners on the other hand are used to adjust component parameters to the given user, and by this personalize the closed loop.

Considering the abstraction level hierarchy, the reflective layer consists of analysis and coordination components providing abstractions from sensor data (i.e., features), and coordinating the operation of multiple actuators. Analysis components create abstract representation of the user's state and his context from available data: among others the user's current mood, cognitive workload, and physical conditions. In the opposite direction, coordination components map high-level plans to operations of – possibly several – actuators. In the abstraction level hierarchy, the application layer consists of components reasoning only on the abstract representation of the user and his context created by the reflective layer components, and sends abstract plans to coordinators in the reflective layer.

In a perfect reflective application, the two hierarchies (reuse and abstraction) will coincide: the analysing and coordinating components will be reusable among several applications, while application-specific components will solely define high-level goals using high-level representations of state. For some reflective applications, however, there is still a gap between both hierarchies. This gap is due to the fact that it is yet not always possible to create a common high-level abstraction of the user state and environment on which application components can reason, while still yielding a satisfactory behaviour [10]. Therefore, the reuse hierarchy allows for separating reflective layer and application layer where both hierarchies diverge. Nevertheless, the ultimate goal to merge reuse and abstraction hierarchy remains.

B. *Reflective Component Model*

Components are the major building elements of the reflective layered middleware. They are used to implement reflective system functionality at any level of abstraction and complexity (e.g., sensor input services, higher level diagnoses, and goal-based reasoning components are all represented by the same structural elements). In order to make a uniform and versatile component-based support, a comprehensive meta model has been designed, programmed in Java language and deployed in practice.

The structure of the reflective component is depicted in Figure 2. The class is declared to be a component by extending the "AComponent" class, and declaring a dependency through required ports. In order to satisfy those dependencies components have to be instantiated and connected with each other. In the configure method (in BundleContainer class), which is called on system start up, the developer can specify creation and connection rules. Connection rules may be either very precise or loose, as required by the system design. The functionality of a reflect entity is encapsulated within the component and function groups are placed within component containers. In order to communicate components can offer or require functionality, depending on whether a component sends or receives data (e.g., sensor services are modeled with components that only provide functionality, i.e., sensors' measurements). A same component may provide multiple functionality, in which case multiple ports are to be used (either different in case of different functionalities or instantiated, in the case of the same functionality). Required functionalities are obtained by connecting to the ports of other components that provide them. Connectors are responsible for tracking and binding the components. Component and bus manager represent core functions of the model, providing the access to other components, connectors and containers. With such a model, encapsulation, clear inter-component communication and a sound software composition are ensured. All these features contribute to faster development, testing and deployment of reflective software. Reflective framework offers all the needed functionality for this component model and serves as its run-time environment.

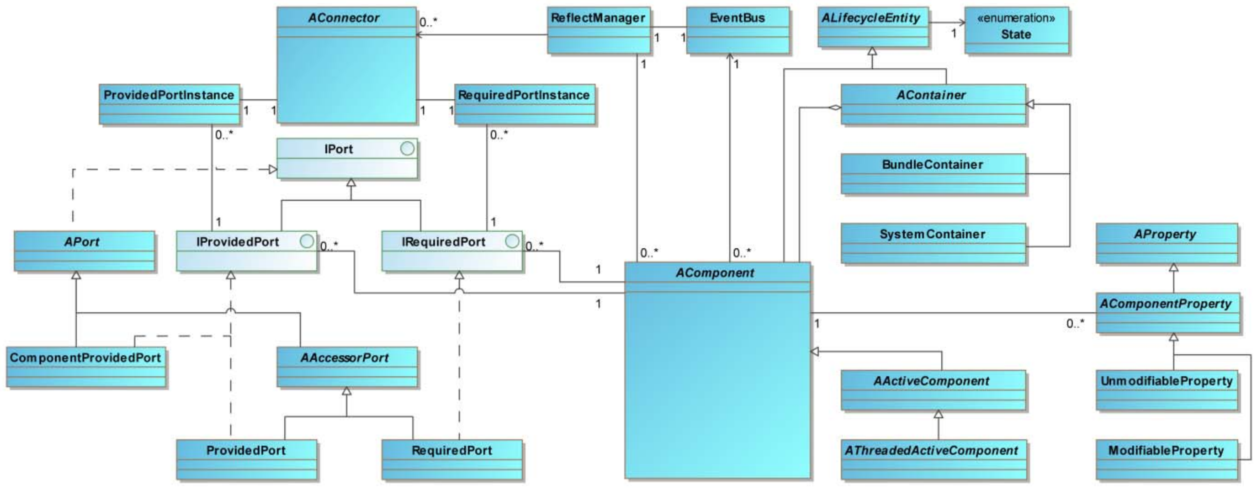


Figure 2. Reflective component model

C. *Implementation Details*

The reflect component model is a Java-based component framework running on top of OSGi [11]. Using OSGi allows to dynamically load and unload Java classes at run-time, adding to the flexibility of the reflective approach. One of the main features of OSGi is the registry mechanism allowing to register objects as services based on interfaces and properties, and performing query services in the same way. In the reflect framework, these facilities are used to add capabilities to the reflect core component framework: user-level OSGi bundles can add component and connector factories, thereby enabling the core framework to create instances of these components and connectors.

Connectors are not predefined by the component framework, but are “fat” user-level entities [12][13]. In this way, a software developer has the flexibility to define the communication patterns between components using the user-defined connector, and the way the communication is affected by reconfiguration (e.g., whether the state of communication must be preserved under reconfiguration, or whether a communication protocol must be ended before reconfiguration may take place).

The reflective component framework defines three types of components:

Passive components offer functionality to other components by declaring one or more provided ports. They might call other components via their required ports, but only do so within the processing of communication received from their own provided ports.

Active components may also provide ports towards other components, but, unlike passive components, they also implement autonomous behaviour. This behaviour is realized by a thread running concurrently to the threads processing the received communication, as well as the threads of other components or the framework. An active component may issue calls to required ports both during processing of calls issued to its own provided ports and in the course of the autonomous loop.

Composite components are containers for other components. A composite component owns an assembly of components and connectors and encloses them with a facade that makes the composite component act like a normal active component. Provided and required ports of the composite component are delegated to compatible ports of enclosed components.

In the reflect component framework, components may additionally declare properties that can be used to query and manipulate its configuration or its state. These properties can be associated with constraints that are automatically enforced by the framework when the property is manipulated. Though it would also be possible to make the component state accessible via provided ports, properties offer a generic and very comfortable way for developing components that offer means for monitoring and adaptation.

Furthermore, the framework offers a message bus facility. Components can register to topics on the bus by method annotations declaring to topic listened to. This message bus facility can be used by components to listen to system events or as a way to easily realize broadcast-style communication between components.

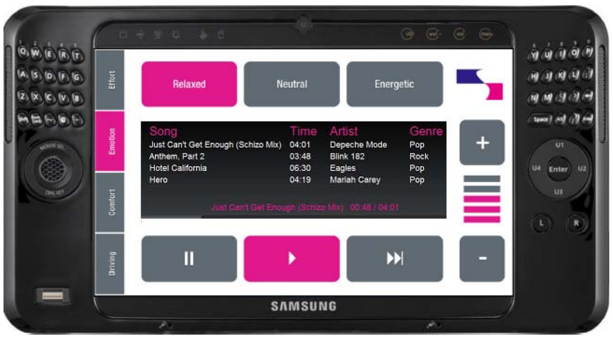
The reflect architecture is realized with the help of OSGi and the reflect component framework defined on top of it. The reflect architecture therefore comes in a set of OSGi bundles.

- A reflect core bundle defining the core concepts.
- A reflect generic bundle defining the generic extensions of the core.
- A reflect ontology bundle containing standardized interfaces and data types from the ontology.
- A set of reflective layer bundles containing reusable components for analysis, coordination, and other purposes.

By registering the reusable components on bundle start-up, they can be created by the reflect component framework. Then, application-specific assemblies can instruct the reflect component framework to create reusable components and connect them with sensors, actuators, and application-specific components.

IV. APPLICATION EXAMPLE

The reflect system has been tested on several prototype demonstrations, home ambient [14], vehicular support [6] and mood player [15].



Song	Time	Artist	Genre
Just Can't Get Enough (Schizo Mix)	04:01	Depeche Mode	Pop
Anthem, Part 2	03:48	Blink 182	Rock
Hotel California	06:30	Eagles	Pop
Hero	04:19	Mariah Carey	Pop

Just Can't Get Enough (Schizo Mix) 00:48 / 04:01

Figure 3. PC as a mood player

The mood player, implemented on Samsung ultra PC as shown on Figure 3, deploys so called "music directs your mood" concept. The psychological background and the control strategy of the music player are described elsewhere [15]. A music player functions as a closed loop repeatedly measuring the current mood state of a user and selecting music from the user's own music database depending on the current mood and a predetermined target mood state. Since a positive mood enhances several cognitive processes, the ability to improve mood is particularly interesting in driving or working situation [15], but also at home in a more relaxed setting.

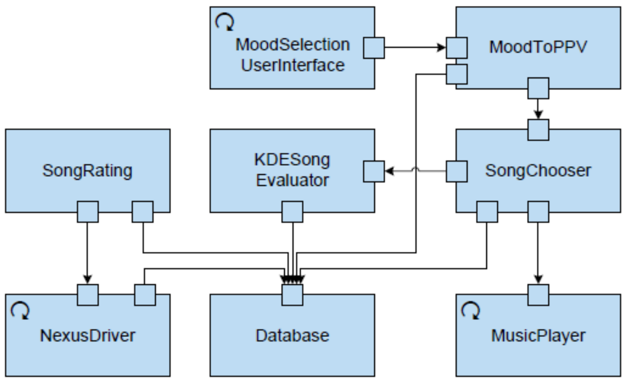


Figure 4. Component model of the mood player

Figure 4 illustrates the component model of the mood player. It embraces eight major components that are connected to each other. The horizontal layout indicates major architectural abstractions: (1) tangible layer (components controlling Nexus driver i.e., sensor devices, database and the actuator i.e., music player); (2) reflective layer (components performing song evaluation, rating and selection) and application layer (components performing user interface and the mood selection).

Table 1 illustrates some of the performance figures and implementation details of the mood player prototype. It can be seen that even on a low-performance computer, the system runs effectively and is flexible for further extensions. To our knowledge, there are no similar systems that could be used for the performance comparison.

TABLE 1: MOOD PLAYER SYSTEM FIGURES

Mood Player		***Comment***
Size (source Java code)	8,6K lines	Extra requirements: 1. Reflect Framework: 28,4K lines and 2. Monitoring tools: 19,4K lines
Run time memory use	30 MB	Most of memory is used by OSGI and external libraries
External devices	Nexus 10	Used for physiological measurements
PU load at an ultra PC	60%	Min. Requirements: Windows XP, 128 MB RAM, 60MB hard disk, TCP/IP
Development Environment	Java 1.6, Eclipse RCP Tool, Equinox OSGi, MySQL, Reflect custom developers tool (for interfacing external devices and rule engine)	

V. CONCLUSION

Developing pervasive adaptive computing applications is still a domain not well understood by classical software engineers. This often leads to poorly engineered and not well defined system structure. The inevitable result is difficult maintenance and poor extensibility of present systems.

To overcome these problems, reflective approach investigates software infrastructures and patterns for pervasive adaptive systems, characterised by seamless integration in the everyday environments. Reflective systems make use of available physical devices to sense and derive the state of their environment and their user, infer the user's current context and conditions, and finally, try to improve the overall users' conditions accordingly.

The paper discussed the requirements and challenges that software engineers have to deal with when implementing the adaptation phenomenon. A generic component framework has been introduced, geared towards ease of use and flexibility. It consists of three layered architecture. The goal of the three layered architecture is to provide the software engineer with prefabricated, generic components for analysis, coordination and dynamic re-configuration. It also encourages early prototyping as many parts of the systems can be simulated [16] by dummy services/components (having the same interface to the rest of the system) and later on substituted by the real parts. The reflect component framework offers an easy-to-use development environment that can be picked up easily by software engineers familiar with Java and OSGi. Reflective approach has been successfully tested in home ambient [7][14], public advertising [7] and vehicular domains [6,7].

The further work is oriented towards improving the communication modules of the reflective framework that should allow for the exchange of the information among different reflective applications in a pervasive manner. Other research topics cross the disciplines, as the techniques for diagnosing different human psychophysiological states need to be further improved, enlarging the application spectrum.

ACKNOWLEDGEMENT

The author expresses his thanks to Dr. Andreas Schroeder from LMU Munich, who read the previous versions of this paper and plaid significant role in design and development of the reflective component model. Most of the work presented here has been done under the REFLECT project [7] (project number FP7-215893) and ASCENS project [17] (project number FP7- 257414), both funded by the European Commission within the 7th Framework Programme.

REFERENCES

[1] D.A. Norman. *The Design of Future Things*. 2007, New York: Basic Books.

[2] A.T. Pope, E.H. Bogart and D.S. Bartolome. Biocybernetic system evaluates indices of operator engagement in automated task. *Biological Psychology*, 40, 1995, 187-195.

[3] S.H. Faircloug. Fundamentals of physiological computing. *Interacting with Computers*, 21, 2009, pp. 133-145.

[4] N.S. Serbedzija and S. Fairclough. Reflective Pervasive Systems. ACM Transactions on Autonomous and Adaptive Systems (TAAS), Vol. 7 (1), April 2012.

[5] G. Kock, M. Ribaric and N. Serbedzija. Modelling User-Centric Pervasive Adaptive Systems - the REFLECT Ontology. In: *Intelligent Systems for Knowledge Management*, Vol. 252. Nguyen, Ngoc Thanh; and Szczerbicki, Edward Eds. Series: "Studies in Computational Intelligence", Springer 2009, ISBN: 978-3-642-04169-3.

[6] N. Serbedzija, A. Calvosa and A. Ragnoni. Vehicle as a Co-Driver, *Proc. 1st Annual International Symposium on Vehicular Computing Systems - ISVCS 2008*, Dublin, Ireland.

[7] REFLECT. REFLECT project - Responsive Flexible Collaborating Ambient, available at, http://reflect.first.fraunhofer.de (2012).

[8] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, 1998.

[9] J.B. Bradbury. Organizing definitions and formalisms of dynamic software architectures. *Technical Report 2004-477*, Queen’s University, 2004.

[10] A. Schroeder, M. Zwaag and M.A. Hammer. Middleware Architecture for Human-Centred Pervasive Adaptive Applications, *Proc.1st PerAda Workshop at SASO 2008*, Venice, Italy, Oct. 21th 2008.

[11] OSGi Alliance. OSGi service platform release 4. http://www.osgi.org/, 2005.

[12] K.K. Lau, P.V. Elizondo and Z. Wang. Exogenous connectors for software components. *8th International SIGSOFT Symposium on Component-based Software Engineering, volume 3489 of Lecture Notes in Computer Science*, Springer, 2005, pp. 90–106.

[13] T. Bures, P. Hnetynka and F. Plasil. Runtime concepts of hierarchical software components. *International Journal of Computer & Information Science*, 2007, 8:454–463.

[14] N. Serbedzija, Reflective Assistance for Eldercare Environments, *SEHC’10, Proceedings of Second Workshop on Software Engineering in Health Care*, Cape Town, May 2010.

[15] J.H. Janssen, E.L. Broek, and J. Westerink. Personalized affective music player. *Proceedings of the 2009 International IEEE Conference on Affective Computing and Intelligent Interaction (ACII),* September 10-12, 2009, Amsterdam, pp. 472-477.

[16] N.S. Serbedzija. MACS – Modular Affective Computing Simulator, *Proc. of Applied Simulation and Modelling*, 2008, June 23 - 25, 2008, Corfu, Greece.

[17] ASCENS. ASCENS project – Autonomic Service Component Ensembles, available at, http://www.ascens-ist.eu/, 2011.