Context Acquisition and Acting in Pervasive Physiological Applications*

Andreas Schroeder, Christian Kroiß, and Thomas Mair

Ludwig-Maximilians-Universität, München, Germany
{schroeda,kroiss}@pst.ifi.lmu.de

Abstract. Physiological computing means using physiological sensors in computing. This is a natural and promising continuation of pervasive computing: as smart devices begin to permeate the environment, they can be used to collect information about the user's emotional, cognitive and physical state to improve the context-awareness of applications. Creating pervasive physiological computing applications is hard, however. We propose a software framework that simplifies the creation of these applications by providing a first design as well as support for processing sensor data, distributing analysis results, and decision making under the uncertainty that arise in physiological computing. We illustrate the presented framework with the personalized affective music player, a context-aware physiological application that plays music to guide the mood of a user into a pre-defined direction.

Keywords: Pervasive Adaptation, Pervasive Computing, Physiological Computing, Software Engineering.

1 Introduction

The next step after pervasive computing [8], that is to say, overcoming desktop-oriented IT environments, is to include physiological information becoming available through the entailed proximity of computer devices to the human. A plethora of physiological sensors exist allowing computers to comprehend the user's emotional, cognitive and physical states. Approaches leveraging these source of inputs are known as physiological computing [1,7], or affective computing [12], when restricted to emotional state. Context-aware interactive applications benefit from physiological computing by gaining a better understanding of their users and thus offering improved services.

Creating pervasive physiological applications is no easy task, however. Data from physiological sensors need to be handled appropriately: signal quality and noise, personal variance, and sometimes just the large amount of data makes this a non-trivial issue. Algorithms and techniques for psychophysiological inference, that is to say, extracting a user's emotional, cognitive and physical state from sensor data are still being researched [7]. Therefore, physiological computing applications need to adapt swiftly to new insights in the domain as well as new emerging algorithms and techniques.

In this paper, we describe how a framework approach can support the development of pervasive (i.e. especially context-aware) physiological applications (Sect. 4). We have

* This work has been partially supported by the EC project REFLECT, IST-2007-215893.

P. Sénac, M. Ott, and A. Seneviratne (Eds.): MobiQuitous 2010, LNICST 73, pp. 393–400, 2012.

designed and implemented the component-based REFLECT framework that supports the software engineers in designing and creating the analysis part (Sect. 4.1) and the decision making part (Sect. 4.2) of a pervasive physiological application, as well as testing and understanding support offered by the accompanying tooling (Sect. 4.3). We use an implementation of the personalized affective music player [9] (Sect. 3) as a running example to motivate the framework support. Finally, we review related work in Sect. 5 and conclude in Sect. 6. First of all however, we start with a discussion of the challenges in pervasive physiological computing (Sect. 2).

2 From Sensing to Acting - A Software Engineering Perspective

From a bird's eye view, a pervasive physiological system is a software agent that interacts with the physical world and the user through sensors and actuators. The agent translates input that it receives through sensors – often called percepts – to actions on actuators that the agent controls [13]. In a pervasive physiological computing setting, the following challenges arise in addition to correctly interpreting percepts: Firstly, the continuous and noisy character of the data data provided by physiological sensors must be handled properly. Secondly, the results of interpreted data must be distributed to other devices in the pervasive computing environment in order to share and combine knowledge to create a more complete picture of the system's physical context. Thirdly, hidden or unknown parameters and incomplete knowledge in the physiological domain may lead to seemingly inconsistent reactions of the environment and the user to actions performed by the system.

Besides the algorithmic challenges, organizational challenges also arise when building pervasive adaptive applications. Software designs for pervasive adaptive applications must support different activities, namely 1) exploring and experimenting with new algorithms and new techniques, 2) testing and validation of the system and its parts, and 3) rapid creation of software artefacts.

Exploration and experimentation and *testing and validation* must be supported since the algorithms that will actually allow to extract knowledge about the emotional, cognitive, and physical state of a human from basic sensors and features are still a field of active research [7]. A software design or software framework must therefore support algorithm exchange and experimentation at low cost. *Rapid creation of software artefacts* is crucial while still being in the process of understanding the application domain and identifying the hard problems within it. Even later, while creating a product, being able to swiftly create the software artefacts may turn out as crucial for reducing the product's time to market.

3 Affective Music Player Example

The affective music player (AMP) is a pervasive physiological application whose concepts stem from [9]. It functions as a closed loop, repeatedly measuring the current mood of the user and selecting music from the user's own music database in order to influence the user towards a user-defined target mood. In this context, mood is understood as a long lasting (i.e. minutes to days) affective state with no clear cause or origin [16].

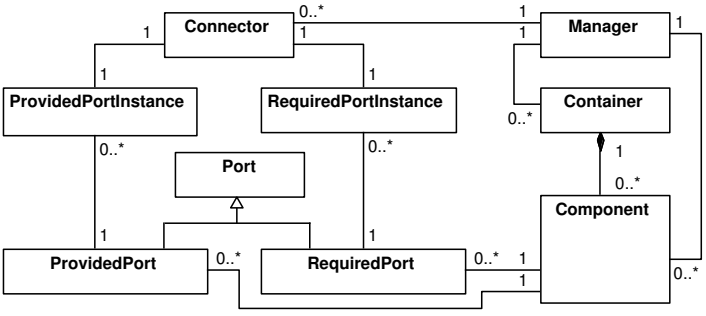

Fig. 1. REFLECT component metamodel

Moods are different from emotions, which are short lasting (i.e. seconds to minutes) affective processes related to an event in the environment or a thought [5].

We built a fully functional prototype of the AMP using the REFLECT framework which is described in the following sections. The AMP serves as a running example to demonstrate the data handling and action selection support the REFLECT framework provides.

4 Framework Support

Handling the challenges described in Sect. 2 without a good design is a daunting task; every step in the process of creating a full system must be carefully thought through from beginning to end with little or no guidance in the design and implementation process. The framework approach followed in the REFLECT framework is one possibility to guide this process.

The basic structure of the REFLECT framework is component-based (see Fig. 1). That is to say, every entity that is created for the REFLECT framework is a component or framed within the context of components: Functionality is encapsulated in *components*, and function groups are arranged within *component containers*. Components communicate to each other over *required* and *provided ports*, which describe the required and provided functionality, respectively. Thus, a component can offer different functionality through different ports; additionally, a component can provide the same functionality multiple times through *multiple instances* of a provided port. Similarly, a required port can collect and bind a set of provided ports. *Connectors* track the binding of ports. Finally, a single *manager* provides access to all components, connectors, and containers. Using components as the basic underlying structure of a system encourages a clear architecture with well-defined interfaces, which is beneficial for rapid development as well as testing.

Fig. 2 shows the basic architecture of applications built on top of the REFLECT framework. Sensors provide transient data that is stored in the context store, that is used by analyser components to extract more abstract features that are stored again in the data context (see Sect. 4.1). The application uses data available in the data context, may make use of persistent data available in the data store (e.g. user preferences), and rater components (see Sect. 4.2) to make decisions on how to control actuators.

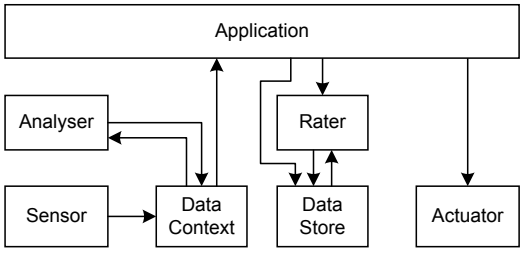

Fig. 2. REFLECT application structure

The REFLECT framework is implemented in Java and built on top of OSGi [2]. It makes use of Java annotations and internal domain specific languages for e.g. container specifications. The implementation itself is focused on ease of use and leverages the Eclipse OSGi tool support [11] wherever possible.

4.1 Handling Sensor Data

Extracting meaningful features from sensor data often requires not only the latest sensor reading, but a set of samples. In the REFLECT framework, this is supported by data windows. These data windows are realized as flexible ring buffer and guarantees to store all recent samples within a fixed time length, and discard old measurements as soon as they exceed the data window time length. Often, a fragment of a data window with specified relative length needs to be created, e.g., to retrieve the last five seconds of data input from a larger data window. The REFLECT framework offers *slices* to create relative fragments of a data window. A slice is defined through start and end points relative to the start and end of the underlying data window. For example, to retrieve the last five seconds of input, a slice is specified as starting at $end - 5s$, and ending at end, where end refers to the end of the underlying data window. Programmatically, start and end are referenced through positive and negative indexes, respectively. Creating a slice with the last five seconds is done programmatically with `window.slice(-5, 0, TimeUnit.SECONDS)`.[1] A slice is live, that is to say, new data added to the underlying data window lets the slice grow and/or shift.

For example, the affective music player infers mood from skin temperature and skin conductance level – two psychophysiological measures known to be related to autonomous nervous system activity [16]. Both skin temperature and skin conductance measurements are stored in data windows in the data context. Both data sources are known to be highly dependent on sensor placement and environment temperature, and in addition, the amplitude of changes in skin temperature and skin conductance have a high interpersonal variation [12]. Therefore, the sensor inputs need to be normalized, which can be easily performed using the data window framework.

[1] Interestingly, zero must reference the data window start point when used as start parameter, and also reference the data window end when used as end parameter. Otherwise, it would be impossible to create slices both starting and ending with the underlying data window.

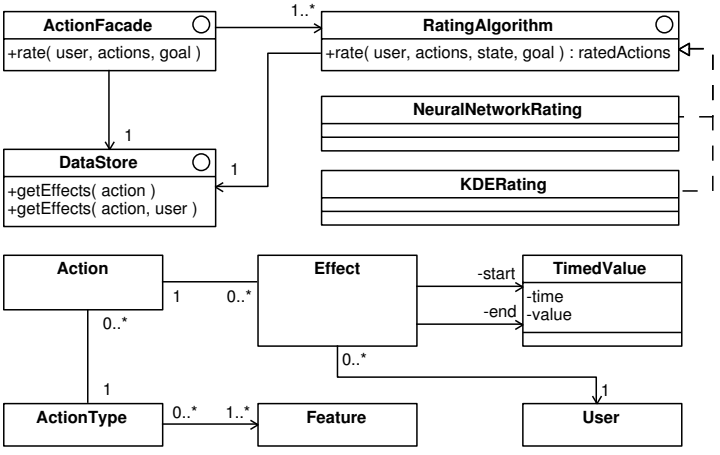

Fig. 3. Rating framework

Including data windows as framework data structure offers several benefits. On the one hand, it avoids the problem of blocking sensor processes when the feature extraction components cannot keep up: As opposed to waiting queues, a data window cannot fill up to its maximum capacity and block the sensor process. Instead, it will always present the most recent data to the feature extraction algorithm using the data window.

In addition to this beneficial process separation, data windows offer a clear concept for distributing data. The process of replicating the content of a data window that is fed by either sensors, feature extraction algorithms or other processes, constitutes an easily understandable and flexible data sharing concept. To put it another way, the process separation offered by data windows allows to distribute processes almost transparently. Within a REFLECT framework instance, a system assembly may specify to replicate the content of a remote data window for local use.

The data context manages all data windows that exist in one framework instance and offers a central point of access. Compared to a connection-oriented design in which a data window may be accessed only by the data producer and its consumers, this design simplifies the access to data windows for new data consumers, data window inspection, and data distribution.

4.2 Selecting Actions

In a pervasive adaptive setting, the problem often arises that the effects of actions are a priori unknown, and need to be learned over time. This is especially true when interacting with humans: reactions may be highly personal, as it is the case with e.g. reactions to music. These highly personal reactions must be caught and learned at run-time. The REFLECT framework offers support for learning action effects and rating actions in two respects: first, it records and stores the effects of executed actions. Secondly, it offers a framework for rating actions based on past effects (see Fig. 3).

In order to record the effects of an action, it must be known a) what part of the environment is affected by the action, i.e. how the action effect can be measured, and b)

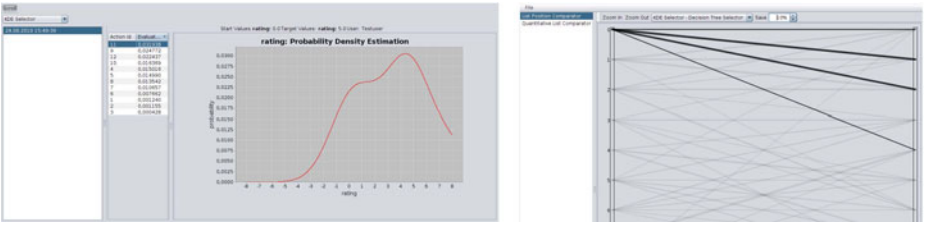

Fig. 4. Tool support for KDE inspection and graphical comparison of rankings

when the action is expected to make an effect. Specifying how action effects are measured is done by grouping actions into action types (e.g. actions consisting of playing a single song are grouped into "a play song action type"), and the affected state features are associated with action types. By specifying only the features that are affected by action types, it is assumed that effects can be tracked uniformly in terms of changes in the affected features. An effect is thereby reduced to pairs of timestamped start and end values. In all applications we created so far, this simplification was feasible. Fig. 3 shows the data model for effects that the framework offers: every effect is associated to a user, as action effects may be highly personal.

Based on the recorded effects in the data store, a rating algorithm can evaluate a set of actions for their fitness to reach a goal state specified by the application. Rating algorithms can be defined using e.g. kernel density estimation techniques (KDE) [14,13] and neural networks [3,13]. The idea in both approaches it to predict the future effect of single actions based on the effects recorded. In the KDE approach, e.g, a Gaussian kernel is fitted on each previously measured effect of an action to create a probability density function for each action. Then, the score of an action is the probability that the action guides the current state into an ϵ-environment around the goal state.

The affective music player uses the action selection framework to select the next song to play once the previous song has ended. It does so on the basis of the past song effects which are recorded in terms of changes in skin conductance level and skin temperature (see Sect. 4.1). In order to select songs, the target mood is brought to desired changes in skin conductance level and skin temperature, and the songs in the user's personalized play list are ranked in terms of their fitness to achieve the requested changes in psychophysiology. The ranking of songs is done using a slight modification of the KDE-based rating algorithm [9].

4.3 Inspecting and Testing

Evaluating the quality of the action selection process can be quite difficult. This is especially true for algorithms like KDE or neural networks, which generally produce results that are hard to interpret. Detailed manual inspection of the internal data and its evolution during simulated runs are often needed. To facilitate this task, the REFLECT framework is accompanied by a tool that provides several visualization options. The tester can use the tool to browse the simulation run and inspect each step, i.e. the updated state of the rating algorithm's internal data and the generated ranking. While the

visualization modules for the internal data have to be designed for each rating algorithm (e.g. KDE, see Fig. 4, left), the visualization tool provides a general module for comparing the action rankings generated by two different algorithms (see Fig. 4, right). It shows a graph consisting of two node columns that represent the aggregated rankings. Edges between the two columns show how actions were ranked differently during the simulation runs. The frequency of rank differences is visualized by different line strength of the edges. This way, the user can easily get a qualitative measure of how similar the action selectors behave.

5 Related Work

Several frameworks and middlewares were developed in the context of pervasive computing. The context toolkit [6] proposes an interesting widget approach for the rapid creation of context-aware pervasive applications. Transparent ad-hoc networking and context information distribution through middleware support is proposed in [17]. The Aura architecture [15] offers a design for transparent content migration that allows content to follow its user.

While most approaches provide better support for transparent distribution of data, all approaches define context in the sense that is understood in pure pervasive computing: the user's physical location, time, devices, their interconnections and proximities, and physical features of the environment such as lighting and temperature (e.g. [6] and [17]). Although some definitions of context are so broad that they can include physiological inputs (especially [6]), the specifics physiological computing brings about are not considered. To the best of our knowledge, no framework or middleware exists that specifically supports the design and implementation of pervasive physiological computing applications.

Rainbow [4] and KX [10] are component-based approaches that use probes and gauges to generate and aggregate streams of measurements for a running application. Then, reconfiguration decisions are made by a controller. The probes, however, are software entities used to provide measurement from legacy systems ([10]) or generic components ([4]) instead of physical sensors.

6 Conclusion

Physiological computing [1,7] is a logical consequence of pervasive computing. Leveraging the physical proximity to the human body offered by smart devices pervading our environment allows to create applications taking into account the user's emotional, cognitive and physical state.

We presented a component-based framework that aims at providing a clear design blueprint, and offering support for processing of physiological data, distributing analysis results, and making decisions in the domain of physiological computing. By this, we aim to simplify the creation of pervasive physiological computing applications.

The REFLECT framework has several limitations, however. First, it does not offer transparent distribution of context information. Instead, each framework instance must

specifically declare its interest in the data provided by a known remote instance. Furthermore, the action rating support currently operates solely on continuous values, and action effects are considered to be measurable by changes in a defined set of features. Validation of the applicability of the framework in other applications than the personalized affective music player is needed. Finally, tool support for automated testing of decision making algorithms is needed.

Looking further ahead, support pervasive physiological computing applications spanning groups of users, from small to large, may also prove a worthwhile extension of the REFLECT framework. Currently, our framework is focussed on supporting single user (or multiple, but sequential users) applications.

References

1. Allanson, J.: Electrophysiologically Interactive Computer Systems. IEEE Computer 35, 60–65 (2002)
2. OSGi Alliance. OSGi Service Platform Release 4.2 (2009)
3. Bishop, C.M.: Neural Networks for Pattern Recognition. Oxford University Press, Oxford (1995)
4. Cheng, S.-w., Huang, A.-c., Garlan, D., Schmerl, B., Steenkiste, P.: Rainbow: Architecture-based self-adaptation with reusable infrastructure. IEEE Computer 37, 46–54 (2004)
5. Damasio, A.D.: Descartes' error: Emotions, reason, and the human brain. Putman, New York (1994)
6. Dey, A.K., Abowd, G.D., Salber, D.: A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. Human-Computer Interaction 16(2), 97–166 (2001)
7. Fairclough, S.H.: Fundamentals of physiological computing. Interacting with Computers 21(1-2), 133–145 (2009)
8. Hansmann, U.: Pervasive computing: the mobile world. Springer, New York (2003)
9. Janssen, J.H., van den Broek, E.L., Westerink, J.H.D.M.: Personalized affective music player. In: 3rd Int. Conf. Affective Computing and Intelligent Interaction. IEEE, New York (2009)
10. Kaiser, G., Gross, P., Kc, G., Parekh, J.: An Approach to Autonomizing Legacy Systems. In: Wshp. Self-Healing, Adaptive and Self-Managed Systems (2002)
11. McAffer, J., VanderLei, P., Archer, S.: OSGi and Equinox: Creating Highly Modular Java Systems. Addison-Wesley Professional, Upper Saddle River (2010)
12. Picard, R.W.: Affective Computing. MIT Press, Cambridge (1997)
13. Russell, S.J., Norvig, P.: Artificial Intelligence: A Modern Approach. Pearson Education, New Jersey (2003)
14. Silverman, B.W.: Density Estimation for Statistics and Data Analysis. Chapman and Hall, London (1986)
15. Sousa, J.P., Garlan, D.: Aura: an Architectural Framework for User Mobility in Ubiquitous Computing Environments. In: 17th World Computer Congress - TC2 Stream / 3rd Conf. Software Architecture, Deventer, Netherlands, pp. 29–43. Kluwer, B.V., Dordrecht (2002)
16. Thayer, R.E.: The biopsychology of mood and arousal. Oxford University Press, New York (1989)
17. Yau, S.S., Karim, F., Wang, Y., Wang, B., Gupta, S.K.S.: Reconfigurable Context-Sensitive Middleware for Pervasive Computing. IEEE Pervasive Computing 1, 33–40 (2002)