An Architecture that Treats Everyday Objects as Communicating Tangible Components

Achilles Kameas
Computer Technology Institute
Achilles.Kameas@cti.gr

Stephen Bellis
National Microelectronics Research Center
sbellis@nmrc.ucc.ie

Irene Mavrommati
Computer Technology Institute
Irene.Mavrommati@cti.gr

Kieran Delaney
National Microelectronics Research Center
kdelaney@nmrc.ucc.ie

Martin Colley
University of Essex
martin@essex.ac.uk

Anthony Pounds-Cornish
University of Essex
apound@essex.ac.uk

Abstract

The paper describes research that has been carried out in "extrovert-Gadgets", a research project funded in the context of EU IST/FET proactive initiative "Disappearing Computer". It presents a set of architectures for the composition of ubiquitous computing applications. The proposed architectures are part of GAS (Gadgetware Architectural Style), a generic architectural style, which can be used to describe everyday environments populated with computational artifacts. The overall innovation of the GAS approach lies in viewing the process where people configure and use complex collections of interacting eGadgets, as having much in common with the process where system builders design software systems out of components. This approach regards the everyday environment as being populated with tens even hundreds of artifacts, which people (who are always in control) associate in ad-hoc and dynamic ways.

1. Introduction

This paper presents a set of architectures for the composition of ubiquitous computing applications. The proposed architectures are part of GAS (Gadgetware Architectural Style), a generic architectural style, which can be used to describe everyday environments populated with computational artifacts.

The paper describes research that has been carried out in "extrovert-Gadgets", a research project funded in the context of EU IST/FET proactive initiative "Disappearing Computer" [3]. This project [4] aims to provide a conceptual and technological framework that will engage and assist ordinary people in (re)configuring or using systems composed of computationally enabled everyday objects, which are able to communicate using wireless networks (these objects are sometimes called "artifacts"). In that sense, artifacts are treated as reusable "components" of a changing everyday environment.

1.1. Basic concepts

The basic definitions underlying this generic concept are:

- *eGadget:* eGadgets (eGts) are everyday physical objects enhanced with sensing, acting, processing and communication abilities. Moreover, processing may lead to "intelligent" behavior, which can be manifested at various levels. eGts can be regarded as artifacts that can be used as building blocks to form eGadgetworlds, with the support of GAS;
- *Plugs:* They are software classes that make visible the eGt's capabilities to people and to other eGts. They may also have tangible manifestation on the eGt's physical interface, so that users can utilize them in forming Synapses;
- *Synapses:* They are associations between two compatible Plugs;
- *eGadgetworld:* An eGadgetworld (eGW) is a dynamic, distinguishable, functional configuration of associated eGts (Figure 1), which communicate and / or collaborate in order to realize a collective function. eGWs are formed purposefully by an actor (user or other) and appear as functionally unified entities.

A *Gadgetware Architectural Style (GAS)* constitutes a generic framework shared by both artifact designers and users for consistently describing, using and reasoning about a family of related eGWs. GAS defines the concepts and mechanisms that will allow people (eGadget users) to define and create eGWs out of eGts, and use them in a consistent and intuitive way.

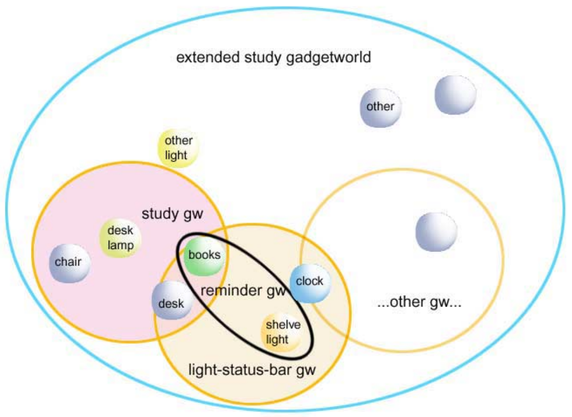

Figure 1. Schematic diagram of the concept of eGadgetworlds

1.2. Innovation

The main idea behind the "ubiquitous computing", or "ambient computing", or "disappearing computer" concept is that the computer "disappears" and computing services are made available to users throughout their physical environment [15].

Several research efforts are attempting to design ubiquitous computing architectures. In the context of the disappearing computer initiative, project "Smart-Its" [10] aims at developing small devices, which, when attached to objects, enable their association based on the concept of "context proximity". Thus, the collective functionality of such a system is mainly composed of the computational abilities of the Smart-Its, without taking into account the "nature" of the participating objects. A more complete and generic approach is undertaken by project "Oxygen", which enables human-centered computing by providing special computational devices, handheld devices, dynamic networks and other supporting technologies [6]. Another interesting disappearing computer project is "Accord", which is focused in developing a Tangible Toolbox (based on the metaphor of a tangible puzzle) that will enable people to easily embed functionality into existing artifacts around the home and enable these devices to be integrated with each other [1].

The overall innovation of the GAS approach lies in viewing the process where people configure and use complex collections of interacting eGadgets, as having much in common with the process where system builders design software systems out of components. This approach regards the *everyday* environment as being populated with tens even hundreds of artifacts, which people (who are always in control) associate in ad-hoc and dynamic ways.

Then, GAS-OS, the software that implements GAS, can be considered as a component framework [14]. GAS-OS manages resources shared by eGts, determines their interfaces and provides the underlying mechanisms that enable communication (interaction) among eGts. Thus, it can be considered as a mini-operating system "residing" between the eGt's intrinsic functions and hardware and the people's will to create more complex behavior.

The proposed concept supports the encapsulation of the internal structure of an eGt (eGts are treated as "black boxes", based on their public interface as manifested by the Plugs), and provides the means for composition of an application, without having to access any code that implements the interface. Thus, this approach provides a clear separation between computational and compositional aspects of an application [14], leaving the second task to ordinary people, while the first can be undertaken by experienced device designers or engineers.

The benefit of this approach is that, to a large extent, system design is already done, because the domain and system concepts are specified in the generic architecture [13]; all people have to do is realize specific instances of the system. The possible variation is declared in the eGt's Plugs, which serve as the primary mechanism for reuse. Composition achieves adaptability and evolution: a component-based application can be reconfigured with low cost to meet new requirements [10], [13].

1.3. Structure of the paper

The rest of the paper is organized as follows: in the next section, the principles that underlie the GAS approach are described. Section 3 presents the architecture of an eGt in different levels, starting from a hierarchy of concepts and moving into the description of a high level functional architecture, a detailed architecture and its implementations. The next section describes the software architecture of GAS-OS. The paper closes with a discussion on the proposed approach, which includes an example scenario of application and a presentation of aspects of emerging behavior, as it is treated in the proposed concept.

2. Principles, perspectives, dimensions and scaling

By nature, eGts exhibit a dual presence, both in the real and cyber (digital) worlds. In the former they appear as tangible objects, occupying physical space for a certain amount of time. In the cyber world, eGts appear as software entities, which are instantiated and "run" on a processing unit. These two "selves" of an eGt are tightly interrelated: every eGt must have a representation in both worlds and changes in one representation may affect the other.

This "form vs. function dualism" is the main driving force behind the "extrovert Gadgets" approach, which gives rise to a set of seemingly competing concepts. The composition of eGts into an eGW is similar to the construction of a software application using software components. Thus, the "extrovert Gadgets" project approaches the problem both from software engineering and design engineering perspectives.

The basic assumption is that an eGt is an autonomous and self-contained artifact, which locally manages its resources (processor, memory, sensors/actuators etc). All eGts are peers and no master eGt needs to be used. Some eGts will be intelligent, in the sense that they will be able to learn and improve their function by observing the consequences of their actions. On the other hand, eGadgetworld construction by lay people has some interesting particularities with direct implications on the design of the GAS:

- In contrast to the typical system engineering process, there is no a priori system to be built. People just try out their ideas until a satisfactory gadget configuration is reached;
- People should not need to be engaged in any type of formal "programming" in order to achieve the desired functions. However, they need to be provided with tools equivalent to a "programmer's workbench", so that they can compose, trace and debug the eGWs;
- eGts are not necessarily provided as all-contained black-box entities, but can also act as "parts" for building larger "wholes". Thus, the principles for deciding on an eGt's level of conceptuall granularity need to be identified;
- eGts need to explicitly "advertise" their interconnection capabilities to users through a comprehensible "vocabulary" and metaphors (e.g. by shape, visual probes, handles, etc).

People and technology are the two dimensions, which also define the scaling possibilities of the concept. In principle, this approach can scale both "upwards" (the assembly of complex, distributed eGts out of simpler ones) and "downwards" (the decomposition of eGts into smaller parts). However, there are limitations imposed both by technology (like number of eGts that can reside within the same wireless network, bandwidth, range, power supply, robustness of sensors and actuators) and people capabilities (like number of eGadgets, Plugs and Synapses, frequency of use, sustainability of eGadgetworlds). The project will investigate the effect of these factors in experiments that will be conducted during the next year.

3. eGadgets as tangible components

An eGt is autonomous, perceived as one entity, though it may have internal structure. It has an ID, a set of Plugs and an internal state, which it manages locally. It can participate in Synapses via its Plugs. It can always state its ID, set of Plugs (and their state) and active Synapses (and their state) in a universally understandable way at a predefined communication channel. Moreover, using a set of intelligent mechanisms, an eGt may be able to locally optimize or adapt its behavior, or even optimize the behavior of an entire eGadgetworld.

A Plug is an abstraction of the properties and abilities of an eGadget. It is implemented as a record and contains attributes and methods, which implement the ways it can be used (protocol), the service it can offer (methods) and its state (attributes). In fact, it is the only way other eGadgets can use eGadget services and have access to the eGadget properties. A Plug is accessible to other eGt modules via GAS-OS only, thus providing a unified way to access the resources of an eGW. All eGadgets come with one TPlug and a set of Splugs (which depends on the set of sensors/actuators of the eGt). The TPlug describes physical properties (one per gadget), while the SPlug describes services (one per service). Plugs have a direct relation to the sensors / actuators and the functions implemented in the eGt by its manufacturer.

Procedurally, an eGW is formed as a set of Synapses. Once a Synapse is established, the involved eGts interact on their own, independently and transparently of the existence of Plugs. An eGW should be considered as being always operational until explicitly disassembled by the user. When switched on, each eGt constantly attempts to re-establish its Synapses. Each Synapse within an eGW may be mandatory (the eGW cannot function without it) or optional (the functionality of the eGW is only reduced without it).

3.1. High-level functional architecture of eGadgets

The above concepts are mapped to a high-level architecture of an eGt (shown in

Figure 2). According to it, an eGadget contains the following software/hardware modules:

- The Gadget Management Software (GadgetOS) is responsible for providing access to the eGt resources (e.g. the RF unit, any sensors or actuators etc);
- The Collaboration Logic provides service discovery services;
- The Computation Logic implements the intrinsic eGt functions.

The following two software modules implement GAS-related services:

- GAS-OS provides plug and synapse management services;
- Agent implements intelligent mechanisms.

This approach follows the standard ISO layer model for network communications and only requires two kinds of interface definitions: GadgetOS to GAS-OS and GAS-OS to Intelligent Agent/Computation Logic.

3.2. eGadget detailed architecture

The tangible self of an eGt is implemented with the architecture shown in Figure 3. An eGt is made of a matrix of sensors and actuators, an FPGA-based board, which implements communication between the sensor or actuator matrix and the processor and a (processor + memory + wireless) module, which is currently served by an iPaq or a Laptop.

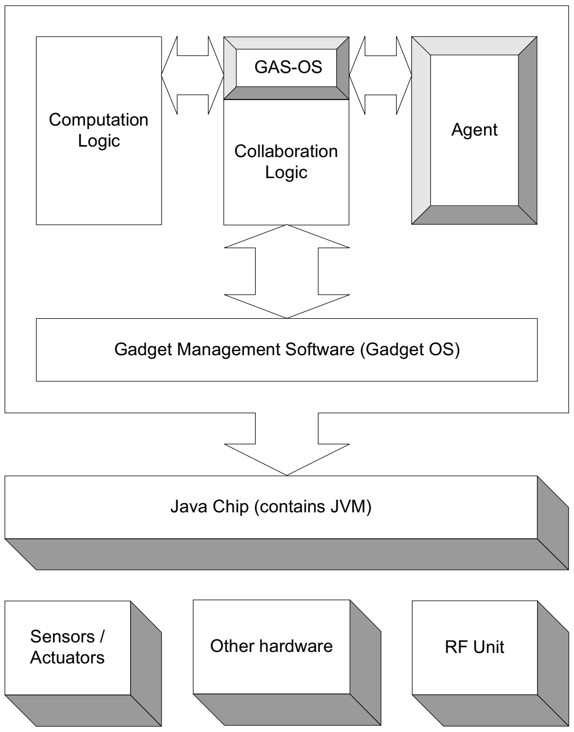

Figure 2. High-level eGadget architecture

The GAS-related middleware includes the following modules: the GadgetOS module, which is specially implemented per every eGt and used to control its resources, the GAS-OS module, which manages the Plugs and Synapses, the eGadget GUI, which at the moment runs as a software simulation and the Agent, which resides on an independent platform and communicates with GAS-OS via sockets.

The communication between the GAS-OS of two eGts is currently implemented using an XML-based messaging system [12].

4. Current Implementation

eGadgets can be broken up into two main classes: sensing eGts and actuating eGts. Examples of artifacts already implemented include eChair, eDesk, eBook, eLamp etc.

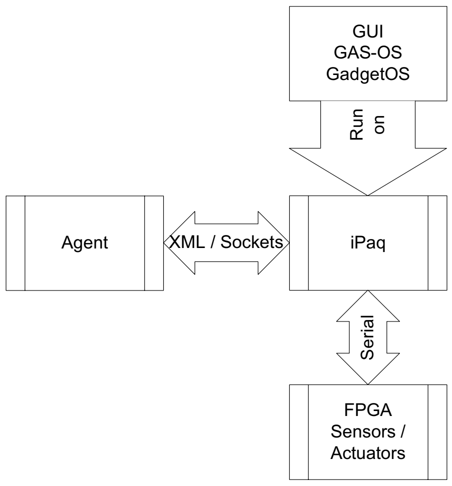

Figure 3. Detailed eGadget architecture

While attempting to implement these eGts, we faced several issues:

- *Networking protocol:* the tradeoff is between a robust, powerful and a lightweight protocol. We decided to use IEEE802.11, partly because it was stable enough at the time of project development and partly because this was not the focus of the project. We are however aware that a lighter protocol, in terms of power and time consumption might be required
- *Messaging layer:* The degree of stability and dependability required from the GAS architecture is beyond any of the P2P implementations that were studied (among them JavaRMI, Jini and Jxta). In addition, since there are numerous platforms and hardware involved, it is not very practical to rely on porting a non-custom architecture. Thus, we decided to have the best of every world by implementing eComP, our own architecture, while adopting some of the excellent work done within the Jxta and other projects.
- *Messaging protocol:* There are a number of advantages of message-based communication over remote procedure calls: it results in more loosely coupled systems, where it is enough for the caller object to create a message understandable by both parties; it is an asynchronous way of communication, so the "caller" does not crash in the case when the server crashes while executing a method; the messages used can contain either text or binary messages wrapped in an envelope, which contains all data necessary to route the message either directly or through intermediate entities that forward the message to the actual receiver without having to look at it; finally, it can be cross-platform.
- *Multicasting:* The peer-to-peer networking module that has been built relies on multicast sockets in order to provide ad-hoc networking capabilities. Peers in the ad-hoc network can be discovered without the

existence of some central registry and routes to them, even multi-hop ones, are discovered on demand. However, this places the requirement on the available wireless network to support multicasting, which is not always the case.

- *Processor and power:* for reasons of portability and robustness, all software is being written in Java. An iPaq, which is equipped with a wireless card and runs the Java Personal Edition is used to provide processing power. The power provided by the iPaq is sufficient for running the software; sensors and FPGAs are powered by separate replaceable batteries.

4.1. Sensing eGadgets architecture

The sensing eGts architecture is based around programmable technology and therefore can be viewed generically as shown in Figure 4.

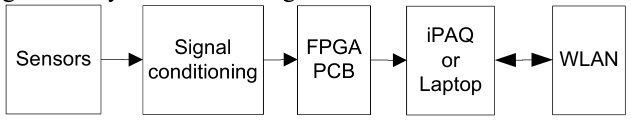

Figure 4: Generic sensing eGadget

The sensors that we have used are pressure pads, light dependent resistors (LDRs), bend sensors, ultrasonic transducers and tilt switches. The pressure pads are used to detect the presence of a substantial weight such as a person sitting on the eChair, while the LDRs have a dual purpose; firstly for detecting a bigger range of weights such as an object on the eDesk and secondly to detect range of light levels, the luminosity of the eBook for example. The eBook also has bend sensors embedded into the spine to detect whether it is open or closed. An ultrasonic transmitter and receiver pair is used to detect the proximity of two objects from each other; the eTable and eChair use this. Signal conditioning logic is used to interface the sensors with a Field Programmable Gate Array (FPGA) based circuit, which does some initial pre-processing on the sensor data before sending it serially to the iPAQ or Laptop. The iPAQ/laptop is used to host the GAS-OS, which interprets the serial sensor data. Wireless local area network PCMCIA cards are then used to interconnect the eGts.

4.2. Actuating eGadgets architecture

The architecture of an actuating eGt (eLamp) is shown in Figure 5. The eLamp receives status information from synapsed eGts over the WLAN and this data is processed by the GAS-OS residing in the iPAQ or the laptop. Information on the required dimmer level is sent serially to the FPGA PCB which implements pulse width modulation (PWM) to produce an analogue signal whose voltage level is proportional to the desired light level. The analogue signal is the input to a voltage-controlled dimmer, which is required to provide the higher voltages necessary to dim the eLamp.

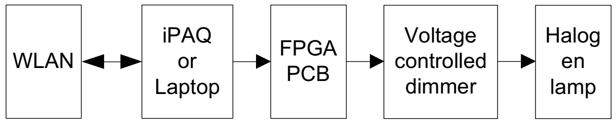

Figure 5: eLamp architecture

4.3. The hardware modules

The hardware modules are based around field programmable gate arrays (FPGAs), which give flexibility for different eGts due to their reprogrammability. The FPGA module, shown in Figure 6, uses the Xilinx Spartan XC2S300E-7FG456C FPGA [8]. This FPGA is relatively low cost while giving high gate equivalent and a large number of digital inputs and outputs. The circuit board includes an on board EEPROM so that the configuration of the FPGA can be retained after the power is turned off. The board also contains fast ZBT SRAM [5] for storage of data if necessary. There are approximately 200 digital IOs available for the input of sensor data, although for the current scenario only a small percentage of these are used, the table representing the maximum usage of 30 LDR inputs. There are also 10 analogue inputs available which use the Texas Instruments TLC549 A to D converter [7].

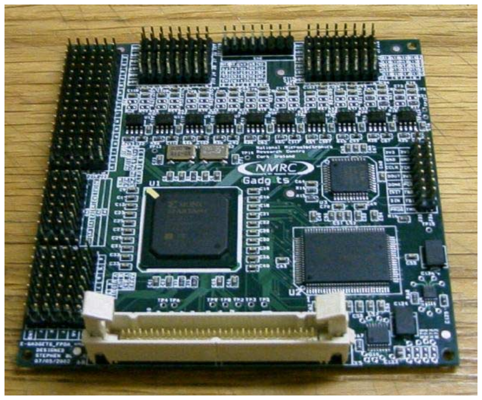

Figure 6: FPGA module

4.4. Interfaces and connectivity

The main interfaces in the eGts are:

- *Sensor → FPGA board.* To implement these interfaces a generic signal conditioning board was designed. This board allows either digital or analogue voltages to be produced from the all the passive sensors that we have used. The circuit includes variable potential dividers, Schmitt trigger drivers and amplification to provide the LVTTL digital signal levels or analogue inputs in the range 0 to 3.3V.
- *FPGA ↔ iPAQ/laptop.* A serial RS232 interface was created for the two-way communication between the

FPGA and iPAQ/laptop. A UART function was programmed onto the FPGA to transfer parallel data into serial format with the required start-stop bits and data ratel. A piggyback board was created to drive the RS232 serial cable from the LVTTL output of the FPGA and buffer the information being sent in the opposite direction. In the iPAQ serial port, Java classes and drivers from serialio.com were used to interface the serial data to the GAS-OS.

- *iPAQ/laptop ↔ WLAN.* PCMCIA expansion packs were used to connect the wireless cards to the iPAQs; these cards could be directly plugged into the laptops.
- *eGt ↔ eGt.* Cisco 350 series WLAN PC Cards were used allow the various eGts to communicate with each other [2]. These cards work on the IEEE802.11b standard, which allows data transfer rates of up to 11Mbps. AD-HOC network mode was used to allow the eGts to communicate with each other without the use of an access point.

5. GAS-OS

GAS-OS is the middleware that runs on every eGt and implements GAS concepts. It offers following services to the user and to other eGts:

- *Plugs discovery and advertising:* The GAS-OS of a specific eGt is responsible for the discovery of all other Plugs (and consequently eGts) within range. For this service, GAS-OS utilizes eComP [12]. It multicasts a *hello message* and all eGts within range respond to it by sending an XML-based advertisement, which contains all the data that one eGt can know about the other, such as the list of SPlugs, the current IP address it has and the port it listens to, etc. Also, the GAS-OS gives the TPlug the ability to give to any connecting Plug the list of SPlugs that an eGt has. Given the fact that every eGt has a TPlug, the GAS-OS guarantees the accessibility of all the eGt Plugs.
- *Synapse establishment – disestablishment:* GAS-OS enables the user to form or destroy Synapses between Plug. It ensures that Plugs get connected only when they are available and "compatible" (an ontology is used to define the degree of compatibility). Then, it takes care of the handshaking between the two connecting Plugs until the Synapse is established. GAS-OS provides the means for successful connection or disconnection, ensuring that these procedures are executed as atomic ones that either succeed or fail before releasing the Plugs. Moreover, it ensures that the Plugs will not stay locked for infinite amount of time, in the case where a Synapse establishment fails. The Plugs are fully functional and do not stay locked during this procedure. This is very important because network delay can be long even for successful Synapse establishments. Also, it is the GAS-OS responsibility to ensure that when an eGadget is shutting down, all connected Plugs are notified. Finally, the GAS-OS ensures that on startup an eGt will attempt to reestablish the Synapses of the eGWs it participated in when it was shutdown.
- *Synapse management:* eGts are notified about changes in the state of other connected eGts via the Plugs. GAS-OS is responsible for sending and handling these notifications and also for forwarding them to the computational logic of the eGt. Thus the GAS-OS acts as a mediator in the eGts collaboration.

5.1. Architecture

GAS-OS software is composed of the following modules (Figure 7):

- *eComP:* it handles the networking communication between the eGts. It is a peer-to-peer software module that enables the dynamic discovery and utilization of remote resources. It is based on message exchanging between remote peers of a network. No infrastructure is assumed to be present, except for TCP/IP networking. Discovery of remote peers is performed using multicast socket connections, thus it does not require some sort of central infrastructure. eComP does not expect the connected peers to be statically bound to a specific IP address. Communication is based on the unique eComP IDs that peers have, which remain the same even when peers may switch networks and utilize new IPs. eComP is implemented for the "J2ME + CDC profile" platform [12].

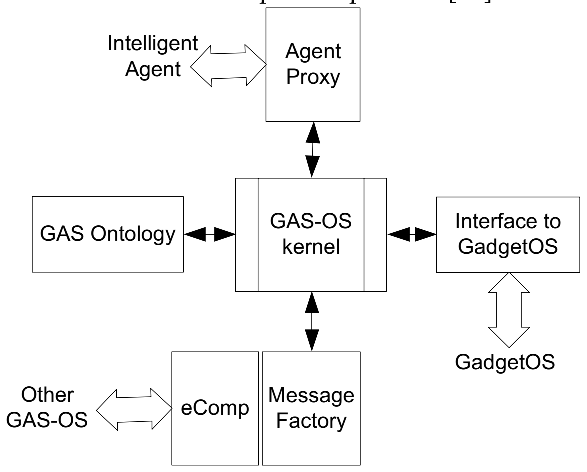

Figure 7. GAS-OS architecture

- *The message factory:* it is responsible for encoding and decoding GAS-OS messages. Since GAS-OS utilizes message-based communication, all data exchange is performed using formatted messages. In order to make the communication procedure easier and to maximize code reuse, this module was developed so that the standard part of XML formatting and decoding that is necessary for the communication is not coded into every single GAS-OS class. The message factory module is implemented for the Personal Java platform.

- *The GadgetOS interface:* it handles communication between the GAS-OS and the GadgetOS of the eGadget. This module offers a standard interface through which the GAS-OS can "talk" to the eGadget specific code that the manufacturer implements. Thus, updating of the eGadget specific code is easy. The critical issue here is that the manufacturer ensures that this code is compatible with Personal Java.
- *The Agent Proxy:* it is the module that is attached to and collaborates with the Intelligent Agent. It is implemented for the Personal Java platform.
- *The GAS Ontology:* it describes the hierarchy of basic concepts and the local eGt capabilities and experience. It is encoded in XML; it contains a set of basic terms that are understandable by all eGts and a mechanism to translate local definitions using the basic terms. It is used to ensure semantic compatibility between different eGts.
- *The GAS-OS kernel:* this is the central module of the GAS operating system. It utilizes the functions of the modules mentioned above to offer the GAS-OS services. The kernel is responsible for initialization and management of Plugs and implements the services offered by the GAS-OS. It has been implemented for the Personal Java platform.

5.2. Intelligent aspects

GAS-OS embeds intelligent mechanisms, some of which have already been implemented with the use of the Intelligent Agent [9]. GAS-OS can provide the following intelligent services:

- *Matching of Plugs:* Synapse formation is based on Plug definition. In some cases, a mapping is formed among the value range of each Plug, as part of Synapse definition. Currently, compatibility is based on data definitions only, but an intelligent mechanism will be developed to automatically deduce compatibility and to propose mappings using Plug description. The GAS ontology will be used as a common referent.
- *Replacement of a missing service in the definition of a GW:* if a Plug is not available at the end of a Synapse (which might mean that the eGt is not itself available), GAS-OS will look for a replacement service offered by another Plug. Again, the GAS ontology will be used to deduce service suitability.
- *Adaptation of mappings:* the user may create an initial mapping for a Synapse between the values of the two participating Plugs. GAS-OS, with the help of the Agent, monitors the way the Synapse is used and adapts the initial mapping to the actual (or changing) requirements of the user.
- *Discovery of usage patterns:* using data gathered by the Agent, GAS-OS can locate patterns in eGts' usage and hence suggest new Synapses.
- *Recording of experience:* an eGt's experience is coded in its internal database of rules.

The Intelligent Agent is a mechanism that handles large vectors of sensor / actuator data. It has access to the local eGt's sensors and to the sensors of other eGts it is connected to via Synapses. The Agent can function both at a local and at an eGW level. In the former, it communicates directly only with GAS-OS, thus ensuring module independence. In the latter, it can work with whatever set of Synapses the user has established and via them will have access to the other eGts.

The Agent forms rules based on the way people use eGadgets and learns new behaviors by modifying the rules. The connected eGadgets form the context in which the Agent produces new rules. So, when the configuration of an eGW changes, the rule set needs updating.

5.3. Interaction architecture

The user interacts with eGts and eGWs with the use of an eGadgetworld Editor (GE). To meet the above requirements, the GE provides the following services to the user:

- Discovery of eGadgets and their Plugs;
- Information about the discovered eGadgets, Plugs, their capabilities and offered services;
- Supporting the user in creation, editing and destroying a Synapse;
- Creation and management of eGadgetworlds.

Two types of GE will be implemented:

- The Full Portable Gadgetworld Editor (FPGE), which runs on a PC or Laptop and offer the complete set of eGW editing functionality
- The Restricted Portable Gadgetworld Editor (RPGE), which will run on a PDA and offer only the basic set of eGW editing functionality (creating and editing Synapses) in accordance to the limited display capabilities of the device

6. Discussion

Let's first assume an example scenario involving the use of eGts. "John, a 21-year-old Law student lives in a student dormitory, in the University campus. He is familiar with PC use, but is by no means a programming expert. John has recently created his Study eGadgetworld, with a new "extrovert-Gadgets" system that he recently bought and has been using for a week. He has set up this eGW to turn on the light automatically, when he is studying on his desk, since the desk light switch is at the

0-7695-1895/03 $17.00 © 2003 IEEE

back of the shelves and it is often a hustle to find it and switch it on”.

The above description refers to a simple eGW composed of the following eGts: Desk, desk-lamp, book, chair. The Plugs of these eGts are shown in Figure 8 (key: O=output plug, I=input plug, H=Higher level (composite) plug). The overall eGW function can be described as:

When the particular CHAIR is NEAR the DESK AND ANY BOOK is ON the DESK AND SOMEONE is sitting on the CHAIR AND The BOOK is OPEN

THEN

ADJUST the LAMP INTENSITY according to the book LUMINOCITY.

Desk	Lamp	Book	Chair
T-plug O	T-plug O	T-plug O	T-plug O
Weight O	On/off I	Open / closed O	Occupancy O
Proximity O	Intensity I	Luminosity O	

Figure 8. The Plugs of example eGts

The Synapses required to implement the eGW are:
Desk (weight=1) -> Lamp (on/off=on)
Chair (occupancy=1) -> Lamp (on/off=on)
Book (open/closed=open) -> Lamp (on/off=on)
Book (luminosity) -> Lamp (intensity)

The support for emerging behavior is a core requirement of ubiquitous computing. In the above simple scenario, John “creates” emerging behavior from the eGts in his possession by associating them in an ad-hoc way. Within the “extrovert Gadgets” project, “emerging behavior” is approached from two perspectives:

- *From the people’s perspective:* people are considered to be active shapers of their environment (which consists of eGadgets), not simple consumers of technology. In project terms, this means that people may form unexpected collections of Synapses by associating compatible Plugs in an ad-hoc way to perform a not predefined function, associating seemingly incompatible or partially incompatible Plugs or using parts of eGadgets via their Plugs in order to form virtual eGadgets
- *From the technology perspective:* GAS technology supports the above actions (thus facilitating emerging behavior) by defining Plugs as abilities, independently of their possible uses, augmenting an eGadgetworld with services found in the environment, creating Synapses based on stated or perceived people goals, maintaining the state of an eGadgetworld and learning from monitoring people behavior

The “extrovert-Gadgets” project is currently halfway along its work plan. The first version of GAS-OS, described in this paper, has been implemented and tested successfully in sample scenarios (for the needs of which, about ten eGts have been constructed). An expert appraisal was carried out to test the basic concepts, yielding valuable insight. Currently, GAS-OS is being redesigned to become a distributed OS and to integrate better intelligent services. During the next year, the formal aspects of the approach will be defined (including the ontology) and experiments have been planned to test the boundaries (imposed by technology and possibly concepts) of the approach.

7. References

[1] Accord project website: http://www.sics.se/accord/home.html

[2] Cisco website: http://www.cisco.com/

[3] Disappearing Computer initiative: http://www.disappearing-computer.net/

[4] e-Gadgets project website: http://www.extrovert-gadgets.net

[5] Micron website: http://www.micron.com/

[6] Oxygen project website: http://oxygen.lcs.mit.edu/

[7] Texas Instruments website: http://www.ti.com/

[8] Xylinx website: http://www.xilinx.com/

[9] Hagras H., et al., *Incremental Synchronous Learning for Embedded-Agents Operating in Ubiquitous Computing Environments*, to appear in the book entitled *Soft Computing Agents: A New Perspective for Dynamic Information Systems*, in the International Series "Frontiers in Artificial Intelligence and Application" by IOS Press.

[10] L.E. Holmquist, F. Mattern, B. Schiele, P. Alahuhta, M. Beigl and H.W. Gellersen, “Smart-Its Friends: A Technique for Users to Easily Establish Connections between Smart Artefacts”, in *Proceedings of UBICOMP 2001*, Atlanta, GA, USA, Sept. 2001.

[11] J. Hopkins, “Component primer”, *Communications of the ACM*, 43(10), pp 27-30.

[12] A. Kameas, D. Ringas, I. Mavrommati and P. Wason, “eComP: an Architecture that Supports P2P Networking Among Ubiquitous Computing Devices”, in *Proceedings of the IEEE P2P 2002 Conference*, Linkoping, Sweden, Sept. 2002.

[13] Meijler, T.D. and O. Nierstrasz, *Beyond objects: components*, in *Cooperative information systems: current trends and directions* (M. Papazoglou and G. Schlageter – eds), Academic Press, 1997

[14] Schneider, J.G. and O. Nierstrasz, *Components, scripts and glue*, in *Software architectures – advances and applications* (J. Hall and P. Hall – eds), Springer-Verlag 1999, pp 13-25.

[15] M Weiser, “Some Computer Science Issues in Ubiquitous Computing”, *Communications of the ACM*, 36(7), pp 75-84.