*Journal of Ubiquitous Computing and Intelligence*
Vol. 1, 1–13, 2006

# A Conceptual Model and the Supporting Middleware for Composing Ubiquitous Computing Applications

Nicolas I. Drossos[1,*], Christos A. Goumopoulos[1,2], and Achilles D. Kameas[1,2]

[1] *Computer Technology Institute, University Campus, Rion, Patras, Greece*
[2] *Hellenic Open University, Greece*

(Received: 7 October 2005; Accepted: Xx Xxxx Xxxx)

Given the resulting complexity of the ambient applications that one can form in the Ubiquitous or Pervasive Computing domain it is required to abstract the intricacies of a heterogeneous supporting environment (e.g., intrinsic characteristics of specific communication models) away from the application logic. These applications will be characterized by the increasing ubiquity of interactions between many possibly heterogeneous artifacts and services. This paper presents the Plug/Synapse abstraction, which provides a conceptual model for building ubiquitous computing applications in a high-level programming manner. GAS-OS is the software layer that implements the Plug/Synapse model and the concepts encapsulated in GAS, a generic architectural style, which can be used to describe everyday environments populated with computational artifacts. The paper examines also the design and architecture of GAS-OS, which is the minimum set of modules and functionalities that every device must afford, in order to be a ubiquitous computing artifact and participate in artifact collections.

**Keywords:** Distributed System, High-Level Programming, Middleware, Modeling, System Architecture, Ubiquitous Computing.

## 1. INTRODUCTION

The vision of Ambient Intelligence (AmI) implies a seamless environment of computing, advanced networking technology, and specific interfaces.[1] In one of its possible implementations, technology becomes embedded in everyday objects such as furniture, clothes, appliances, vehicles, roads and smart materials, and people are provided with the tools and the processes that are necessary in order to achieve relaxing interactions with this environment. The AmI environment can be considered to host several Ubiquitous Computing (UbiComp) applications, which make use of the infrastructure services provided by the environment and the services provided by the objects therein.

An important characteristic of AmI environments is the merging of physical and digital space (i.e., tangible objects and physical environments are acquiring a digital representation). As the computer disappears in the environments surrounding our activities, the objects therein become augmented with Information and Communication Technology (ICT) components (i.e., sensors, actuators, processor, memory, wire-less communication modules) and can receive, store, process, and transmit information; in the following, we shall use the term "artifacts" for this type of augmented objects.

These objects may be new or improved versions of existing objects, which by using the ambient technology, allow people to carry out novel or traditional tasks in unobtrusive and effective ways. The provision of conceptual models and software mediators for creating, managing, communicating with, and reasoning about, these new ecologies (or UbiComp applications) is of paramount importance, because people involvement is considered crucial for the successful adoption of this new computing paradigm.

This paper examines specifically the Plug/Synapse model and the GAS-OS middleware. The former provides a conceptual model for building UbiComp applications in a high-level programming manner. We argue that this model can be comprehended and applied via supporting tools both by developers and users to manipulate at different levels of detail the characteristics of UbiComp applications. In our approach applications consist of

*Author to whom correspondence should be addressed.
Email: ndrossos@cti.gr

 1555-1326/2006/1/001/013 doi:10.1166/juci.2006.204

interacting tangible objects (artifacts), which carry the ICT technology required. The GAS-OS middleware is coupled with these artifacts and supports their collective operation transparently to the user.

The conceptual model and the accompanying middleware are part of the Gadgetware Architectural Style (GAS),[2] which constitutes a generic framework, shared by users and designers, for consistently describing, using, reasoning about UbiComp applications within the AmI environment. GAS proposes the appropriate design vocabulary, configuration, and semantic interpretation rules that facilitate the development of UbiComp applications. GAS considers the process where people configure and use complex collections of interacting artifacts, as having much in common with the process where system builders design software systems out of components. In the proposed approach, the Plug/Synapse model provides a high-level abstraction of the component interfaces and the composition procedure.

The rest of the paper is organized as follows. Section 2 discusses the motivation behind the specification of a conceptual framework for composing UbiComp applications, presents the Plug/Synapse model in a formal way and examines its use through an example of everyday life scenario. The design and architecture of the system software that implements and validates the model is described in Section 3. Section 4 presents some implementation details based on the scenario previously introduced and provides a performance evaluation of the system. Related approaches and work are presented in Section 5. A discussion on the approach presented regarding the end-user empowerment for handling effectively intelligent environments as well as on issues and problems that may increase the complexity of the assembled systems is given in Section 6. Section 7 concludes this paper by presenting final statements and future work.

## 2. THE CONCEPTUAL FRAMEWORK

### 2.1. Motivation

For the AmI vision to succeed, nobody should be excluded from using UbiComp technology or accessing UbiComp system services. Technology must be assisting rather than disturbing people, because they cannot afford to be consumed in learning how to "treat" technology.[3] Thus, on our way to realizing the AmI vision, together with the realization of ubiquitous computing technology, we need a conceptual framework that will bridge the gap between system design and use. This model must be comprehensible both by developers and end users so that the latter are enabled to actively shape the ubiquitous computing environments they live in. Moreover, the visibility of the functionality of the UbiComp system must be controllable by people; people must remain "in the loop," so that they build trust on the system.

The proposed framework carries along the basic technological concepts that allow for inter-associations of objects (this can include a basic set of terminology and supporting mechanisms in order to do this manipulation). To bring these properties in the realm of UbiComp applications, the basic concepts and elements of the component model need to be expressed in a way that they can be easily communicated to people, thus achieving a controlled degree of visibility into the—otherwise invisible—workings of a ubiquitous environment.

In fact this model acts as a high level interface for the user within a ubiquitous computing environment. It becomes a communication medium, which people can perceive, and by having access to it they can manipulate the 'disappearing computers' within their environment. To support people task models, we have adopted the "jigsaw" metaphor, a widespread universal paradigm of interconnectivity.[4, 5]

The principles underlying the proposed conceptual framework are:

*Self-representation*: the digital representation of artifact's physical properties is in tight association of its tangible self.

*Functional autonomy*: artifacts function independently of the existence of other artifacts.

*Composeability*: artifacts can be used as building blocks of larger and more complex systems.

*Changeability*: artifacts that possess or have access to digital storage can change the digital services they offer.

Up to now, the ways that an object could be used and the tasks it could participate in have usually been determined by its shape. Artifacts overcome this limitation by producing descriptions of their properties, abilities and services in the digital space, thus becoming able to improve their functionality by participating in compositions, learning from usage, becoming adaptive and context aware, etc.

The research hypothesis is that even if an individual artifact has limited functionality, it can achieve more advanced behavior when grouped with others. Then the aim is to look at how collections of artifacts can be configured to work together in order to provide behavior or functionality that exceeds the sum of their parts.

### 2.2. The Plug/Synapse Model for Composing UbiComp Applications

Our approach regards the everyday environment consisting of a multitude of artifacts, which people combine and recombine in *ad-hoc*, dynamic ways. By providing uniform abstractions and a supporting middleware, we treat objects as components of a UbiComp application. Artifacts can be considered as information appliances[6] extended with composeability. Each artifact possesses a digital representation of its properties, which it makes available to other artifacts. Based on these representations, artifacts can

be associated in order to achieve synthetic functionality. People are given 'things' with which to make 'new things.' The behavior of these 'new things' (i.e., UbiComp applications) is neither static, nor random, because it is guided by how applications are to be used.

The basic definitions encapsulated in our conceptual framework are:

*Artifacts*: An artifact is a tangible object which bears digitally expressed properties; usually it is an object or device augmented with sensors, actuators, processing, networking unit, etc., or a computational device that already has embedded some of the required hardware components. Software applications running on computational devices are also excessively considered to be artifacts. Examples of artifacts are furniture, clothes, air conditioners, coffee makers, a software digital clock, a software music player, etc.

*Artifact compositions*: Two or more artifacts (simple or composite) can be combined in an artifact composition. Such compositions are the tangible bearers of *UbiComp applications* and are regarded as service compositions; their realization can be assisted by end-user tools.

*Properties*: Artifacts have properties, which collectively represent their physical characteristics, capabilities, and services. A property is modeled as a function that either evaluates an artifact's state variable into a single value or triggers a reaction, typically involving an actuator. Some properties (i.e., physical characteristics, unique identifier) are artifact-specific, while others (i.e., services) may be not. For example, attributes like *color/shape/weight* represent properties that all physical objects possess. The service *light* may be offered by different objects. A property of an artifact composition is called an *emergent* property. All of the artifacts properties are encapsulated in a *property schema* which can be send on request to other artifacts, or tools (e.g., during an artifact discovery).

*Functional schemas*: An artifact is modeled in terms of a functional schema: $F = \{f_1, f_2, \ldots, f_n\}$, where each function $f_i$ gives the value of an observed property $i$ in time $t$. Functions in a functional schema can be as simple or complex is required to define the property. They may range from single sensor readings to rule-based formulas involving multiple properties, to first-order logic so that we can quantify over sets of artifacts and their properties.

*State*: The values for all property functions of an artifact at a given time are the state of the artifact. For an artifact $A$, the set $P(A) = \{(p_1, p_2, \ldots, p_n) | p_i = f_i(t)\}$ represents the state space of the artifact. Each member of the state vector represents a *state variable*. The concept of state is useful for reasoning about how things may change. Restrictions on the value domain of a state variable are then possible.

*Transformation*: A transformation is a transition from one state to another. A transformation happens either as a result of an internal event (i.e., a change in the state of a sensor) or after a change in the artifact's functional context (as it is propagated through the synapses of the artifact).

*Plugs*: Plugs are the constructs that we use to represent properties of artifacts in the digital space. Plugs are characterized by their direction and data type. Plugs may be output (O) in case they manifest their corresponding property (e.g., as a provided service), input (I) in case they associate their property with data from other artifacts (e.g., as service consumers), or I/O when both happens. Plugs also have a certain data type, which can be either a semantically primitive one (e.g., integer, boolean, etc.), or a semantically rich one (e.g., image, sound, etc.). In this paper, only primitive data types are considered. From the user's perspective, plugs make visible the artifacts' properties, capabilities, and services to people and to other artifacts.

*Synapses*: Synapses are associations between two compatible plugs. In practice, synapses relate the functional schemas of two different artifacts. When a property of a source artifact changes, the new value is propagated through the synapse to the target artifact. The initial change of value caused by a state transition of the source artifact causes finally a state transition to the target artifact. In that way, synapses are a realization of the functional context of the artifact.

To achieve collective desired functionality, one forms synapses by associating compatible plugs, thus composing applications using artifacts as components. Two levels of plug compatibility exist: Direction and data type compatibility. According to direction compatibility output or I/O plugs can only be connected to input or I/O plugs. According to Data type compatibility, plugs must have the same data type to be connected via a synapse. However, this is a restriction that can be bypassed using value mappings in a synapse (Fig. 2). No other limitation exists in making a synapse. Although this may mean that meaningless synapses are allowed, it has the advantage of letting the user create associations and cause the emergence of new behaviours that the artifact manufacturer may have never thought of. Meaningless synapses can also be seen as having much in common with runtime errors in a program, where the program may be compiled correctly but does not manifest the desired by the programmer behavior.

The use of high-level abstractions, for expressing such associations, allows the flexible configuration and reconfiguration of UbiComp applications. It only requires that artifacts are able to communicate and they have to run the GAS-OS middleware in order to "comprehend" each-other, so that people can access their services, properties, and capabilities in a uniform way. People in that way would not need to be engaged in any type of formal "programming" in order to achieve the desired functions.

Our model supports three abstraction levels:

*Network independence*: the Plug/Synapse model is independent of the underlying protocols, needed for example to route messages or to discover resources in realization of an application.

*Physical independence*: the services offered by an artifact are independent of the artifact itself; this does not hold for its physical characteristics. Thus the creation of artifact compositions does not require the continuous presence of an artifact (provided they do not involve physical characteristics).

*Semantic independence*: the description of artifact compositions or applications is based only on the types of the participating plugs and is independent of the way the plugs are realized in each artifact.

The approach adopted is that people live in an environment populated with artifacts; they have a certain need or task, which they think can be met or carried out by (using) a combination of services and capabilities; then, they search for artifacts offering these services and capabilities as plugs; they select the most appropriate ones and combine the respective plugs into functioning synapses; if necessary, they manually adapt or optimize the collective functionality.

### 2.3. The End-User's Perspective: Composing a Real-Life Home Application

In the near future, people will be living and carry their activities and tasks in an environment populated with artifacts. As is the usual case, in order to carry these activities out, people will look for services or objects they can use. Using our approach, people will be able to carry out their activities using artifact combinations. All they need to do is mentally decompose the activity into tasks which can be supported by simple services and look for the artifacts that have appropriate properties. Finally, they select the most appropriate ones and combine the respective plugs into functioning synapses; if necessary, they can manually adapt or optimize the collective functionality. Because the composition of artifacts is regarded as a high-level programming task, "run-time" errors may appear causing artifact compositions not to function properly as expected. Optimization is a trial-and-error process: people adapt the synapses and the mappings in order to achieve the desired functionality. To support them in this process and to hide the complexity of artifact interactions, we have developed user friendly tools that implement the conceptual framework (Figs. 1 and 2).

These concepts can be better illustrated if we consider the *Study application* example, which we'll follow throughout this paper. Let's take a look at the life of Patricia, a 27-year old single woman, who lives in a small apartment near the city centre and studies Spanish literature at the Open University. A few days ago she passed by a store, where she saw an advertisement about these new augmented artifacts. Pat decided to enter. Half an hour later she had given herself a very unusual present: a few furniture pieces and other devices that would turn her apartment into a smart one! On the next day, she was anxiously waiting for the delivery of an eDesk (it could

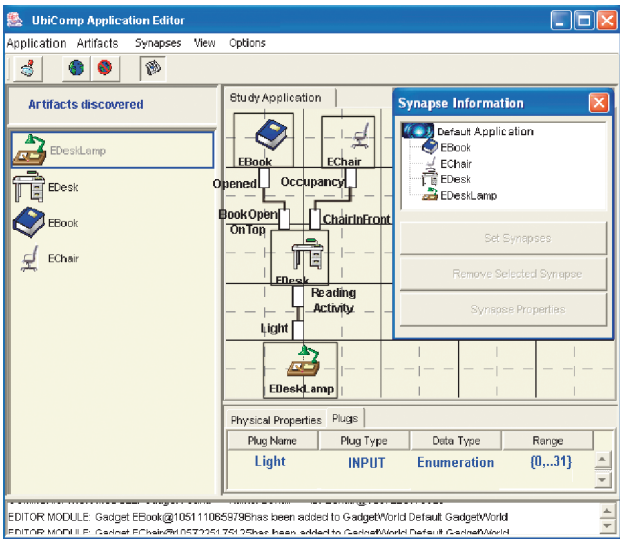


**Fig. 1.** Combined artifacts in the UbiComp application editor.

sense objects on top, proximity of a chair), an eChair (it could tell whether someone was sitting on it), a couple of eLamps (one could remotely turn them on and off), and some eBook tags (they could be attached to a book, tell whether a book is open or closed). Pat had asked the store employee to pre-configure some of the artifacts, so that she could create a smart studying corner in her living room. Her idea was simple: when she sat on the chair and she would draw it near the desk and then open a book on it, then the study lamp would be switched on automatically. If she would close the book or stand up, then the light would go off.

The behavior requested by Pat requires the combined operation of the following set of artifacts: eDesk, eChair, eDeskLamp, and eBook. The properties and plugs of these artifacts are shown in Table I and are manifested to Pat via the UbiComp Application editor tool,[7] an end-user tool that acts as the mediator between the Plug/Synapse conceptual model and the actual system. Using this tool Pat can combine the most appropriate plugs into functioning synapses as shown in Figure 1.

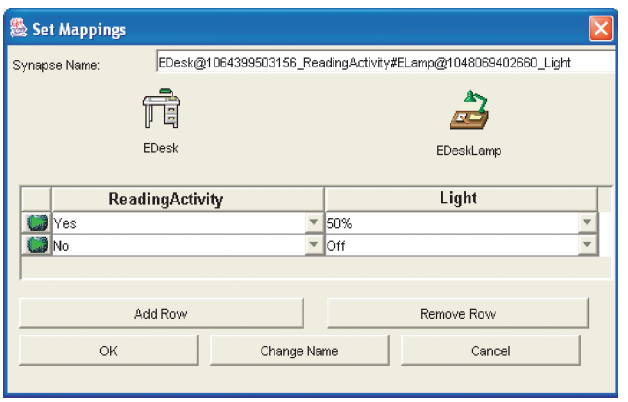


**Fig. 2.** Setting mappings between the eDesk.ReadingActivity and eDeskLamp.Light plugs.

**Table I.** Analyzing the UbiComp application.

| Artifact | Properties | Plugs | Functional schemas |
|---|---|---|---|
| eChair | —Sensing chair occupancy capability (C1)<br>—Transmitting object type capability (C2) | Occupancy: {OUT\|Boolean} | eChair.C1 ← read(pressure-sensor)<br>eChair.C2 is an attribute<br>Occupancy ← {eChair.C1, eChair.C2} |
| eBook | —Sensing open/close capability (C1)<br>—Transmitting object type capability (C2) | Opened {OUT\|Boolean} | eBook.C1 ← read(bend-sensor)<br>eBook.C2 is an attribute<br>Opened ← {eBook.C1, eBook.C2} |
| eDesk | —Sensing objects on top capability (C1)<br>—Sensing proximity of objects capability (C2) | —BookOpenOnTop: {IN\|Boolean}<br>—ChairInFront: {IN\|Boolean}<br>—ReadingActivity: {OUT\|Boolean} | eDesk.C1 ← read(RFID-sensor)<br>eDesk.C2 ← read(proximity-sensor)<br>IF eDesk.C1 = eBook.C2 AND eBook.C1 = TRUE<br>THEN BookOpenOnTop ← TRUE ELSE BookOpenOnTop ← FALSE<br>IF eDesk.C2 = TRUE AND eChair.C1 = TRUE<br>THEN ChairInFront ← TRUE ESLE ChairInFront ← FALSE |
| eDeskLamp | Light service (S1) | Light: {IN\|Enumeration} | IF BookOpenOnTop = TRUE AND ChairInFront = TRUE THEN<br>ReadingActivity ← TRUE ELSE ReadingActivity ← FALSE<br>IF eDesk.ReadingActivity THEN S1(on) ELSE S1(off) |

The properties, plugs, and functional schemas of each artifact participating in the study application.

In the case of the synapse between eDesk.Reading Activity and eDeskLamp.Light plugs, a data type compatibility issue arises. To make the synapse work, Pat can use the UbiComp Editor to define mappings that will make the two plugs collaborate, as shown in Figure 2.

The definition of the functional schemas of the artifacts, that is the internal logic that governs the behavior of each artifact either when its state changes or when a synapse is activated are predefined by the artifact developer (for our example, they are shown in Table I). Rules that require identification of the remote artifact, can be specified using the property schema information which is available in the representation of each of the two artifacts that participate in a synapse.

The eBook, eChair, and eDesk comprise an artifact composition whose emergent property is manifested via the ReadingActivity plug. This plug allows the connection of this composition to other artifacts or compositions. Any artifact composition can be edited to extend the functionality of the application. For example, consider that Pat also buys an eClock and wants to use it as a 2 hour reading notification. The eClock owns an alarm plug that when activated, via a synapse, counts the configurable number of hours and then rings the alarm. To implement her idea, what Pat has to do is to use the UbiComp Application editor to create a synapse between the ReadingActivity plug of the eDesk and the alarm plug of the eClock and specify the number of hours in the Properties dialog box of the eClock.

## 3. GAS-OS MIDDLEWARE

Within an AmI environment, a UbiComp application may be composed of a number of heterogeneous artifacts or devices, which may be stationary or portable. Those artifacts and devices have different, dynamically changing capabilities and ways to use them; yet, all of them can communicate. We also assume that no specific networking infrastructure exists, thus *ad-hoc* networks are formed. The physical layer networking protocols used are highly heterogeneous ranging from infrared communication over radio links to wired connections. Since every node serves both as a client and as a server (devices can either provide or request services at the same time), communication between artifacts can be considered as Peer-to-Peer (P2P).[8]

To cope with heterogeneity and provide a uniform abstraction of artifact services and capabilities we have introduced the GAS-OS middleware that abstracts the underlying data communications and sensor/actuator access components of each part of a distributed system, so that a UbiComp application appears as a single integrated computing facility. GAS-OS follows the Message-Oriented Middleware (MOM) approach, by providing non-blocking message passing and queuing services. Furthermore, to handle the need to adapt to a broad range of devices, we have adapted ideas from micro-kernel design[9] where only minimal functionality is located in the kernel, while extra services can be added as plug-ins.

The use of Java as the underlying platform of the middleware decouples GAS-OS from typical operations like memory management, networking, etc. Furthermore, it facilitates the deployment on a wide range of devices from mobile phones and PDAs to specialized Java processors.

The combination of the Java platform and the GAS-OS middleware, hide the heterogeneity of the underlying artifacts, sensors, networks, etc., and provides the means to create large scale systems based on simple building blocks. The next two sections present the role of GAS-OS into the

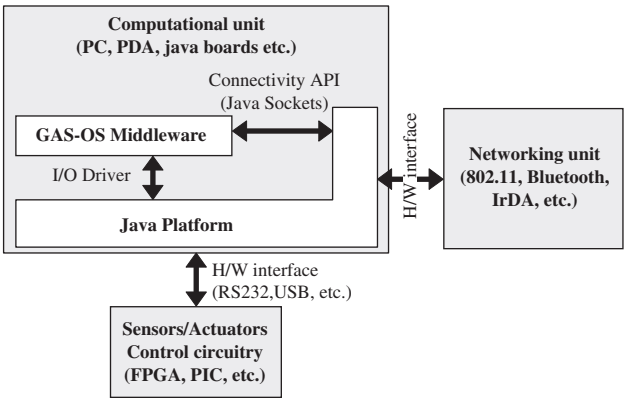


**Fig. 3.** Artifact high-level architecture.

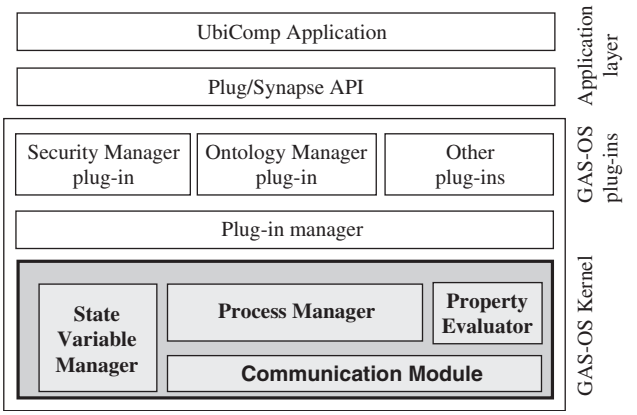


**Fig. 4.** GAS-OS modular architecture.

high level design of an artifact, as well as the architecture of GAS-OS.

### 3.1. The Role of GAS-OS

Figure 3 shows the high-level architecture of an artifact.

A Sensor/Actuator network together with custom control circuitry (e.g., FPGA, PIC micro-controller based boards) are responsible for converting artifact data (e.g., pressure, luminosity, etc.) to digital ones and vice versa. In the case of electronic devices, this circuitry is usually embedded in the device. In both cases digital data are exported to a computational unit using proper hardware interfaces (e.g., RS232, USB, etc.). Although digital data are acquired from the hardware in a uniform way, via the Java Platform, they may be analyzed and handled in a different way for each artifact; thus, a specific GAS-OS I/O driver must be implemented per artifact. A GAS-OS driver is a set of routines linked into the kernel, which are used as part of the mechanism to operate a specific hardware module. Separating the responsibilities of the driver from the middleware facilitates the addition of new hardware modules without changing the middleware. In the same way, via proper h/w interfaces and the Java platform, data from other artifacts are available to the GAS-OS middleware via the Connectivity API.

The I/O driver and connectivity interfaces administer the communication intricacies (e.g., sensor device communication protocols, routing protocols, etc.) in terms of the sensory and interaction communication views respectively of the artifact. Thus, we achieve technology independence and adaptability.

In our approach, an application is realised through the cooperation of artifacts in the form of established logical communication links (synapses) between artifacts acting either as services providers or as service consumers, where services are being manifested through plugs. The GAS-OS kernel implements the concepts of the Plug/Synapse model and the mechanisms to support the composition of UbiComp applications, as explained in more detail in the following section.

### 3.2. GAS-OS Architecture

The outline of the GAS-OS architecture is shown in Figure 4. The GAS-OS kernel is designed to support only accepting and dispatching messages, managing local hardware resources (sensors/actuators), and implementing the Plug/Synapse interaction mechanism. The kernel is also capable of managing service and artifact discovery messages in order to facilitate the formation of the proper synapses.

The GAS-OS kernel encompasses a P2P Communication Module, a Process Manager, a State Variable Manager, and a Property Evaluator module as shown in Figure 4. The P2P Communication Module is responsible for application-level communication between the various GAS-OS nodes. This module translates the high-level requests/replies into messages and by using low-level networking protocols it dispatches them to the corresponding remote peers. The Process Manager is the coordinator module of GAS-OS. Some of its most important tasks are to manage the processing policies, to accept and serve requests set by the other modules of the kernel or to initiate reactions in collaboration with other modules, tasks which collectively serve the realization of the Plug/Synapse model. Furthermore, it is responsible for handling the memory resources of an artifact and caching information of other artifacts to improve communication performance when service discovery is required. The State Variable Manager handles the runtime storage of artifact's state variable values, reflecting both the hardware environment (sensors/actuators) at each particular moment (primitive properties), and properties that are evaluated based on sensory data and P2P communicated data (composite properties). The Property Evaluator is responsible for the evaluation of artifact's composite properties according to its Functional Schema. In its typical form the Property Evaluator is based on a set of rules that govern artifact transition from one state to another. The rule management can be separated from the evaluation logic by using a high-level rule language and a translator that translates high-level rule

specifications to XML that can be exploited then by the evaluation logic.

Following a layered modular architecture allows the replacement of a module without affecting the functionality of the rest provided that the APIs between them remain consistent. This principle holds for the different layers of the architecture as well as within each layer. The modular design of GAS-OS, for example, allows the integration of up-to-date algorithms and protocols in the form of plug-in modules.

Extending the functionality of the GAS-OS kernel can be achieved through plug-ins, which can be easily incorporated to an artifact running GAS-OS, via the plug-in manager. Using ontologies and the ontology manager plug-in all artifacts can use a commonly understood vocabulary of services and capabilities, in order to mask heterogeneity in context understanding and real-world models.[10] In that way, high-level descriptions of services and resources independent of the context of a specific application are possible, facilitating the exchange of information between heterogeneous artifacts as well as the discovery of services. The security manager plug-in on the other hand, when developed, will be responsible for realizing the security policies of each artifact. These policies will be encoded as rules in the ontology, thus becoming directly available to the Process Manager. The security manager will mediate information exchange via synapses in order to ensure that security policies are respected.

## 4. IMPLEMENTATION

The current version of GAS-OS has been implemented in the Java Personal Edition (PE) that is fully compatible with the Java Standard Edition 1.1.8. So far, GAS-OS has been tested in laptops, IPAQs and finally in the EJC (Embedded Java Controller) board EJC.[11]

The following sections will describe implementation details concerning three basic functions supported by GAS-OS in order to realize ubiquitous computing applications using the example introduced in Section 2. First, synapse management, a mechanism that handles the process of establishing logical channels (synapses) among artifacts, then inter-artifact communication, a mechanism that supports the formation and operation of synapses at the network layer by establishing peer-to-peer connections over the physical layer, and finally, the hardware management mechanism, which describes how GAS-OS handles the sensors and actuators of an artifact in order to satisfy the high-level behavior dictated by the association with other artifacts.

### 4.1. Synapse Management

The management of synapses is performed by the Process Manager module. The Process Manager collaborates with the Communication Module and the State Variable Manager (Fig. 4), and sets up an event based internal messaging system that combines input from sensors and actuators with input received from other artifacts, via the network.

As an example, let's consider the synapsing process among the ReadingActivity plug of the eDesk and the Light plug of the eLamp:

*Synapse request*: Synapse request occurs after an artifact has discovered a second one, thus property schemas of each artifact are available to each other. The eDesk sends a "Connection Request" message to the eLamp. The message contains information concerning the eDesk and its ReadingActivity plug as well as the name of the Light plug.

*Synapse response*: When the eLamp receives the message it first checks the plug compatibility of the ReadingActivity and Light plugs. In the example, the Reading plug is output and the Light plug is input, so the direction compatibility test is passed. Data type incompatibility does not hapt the synapsing process, however it needs to be dealt via the use of mappings. Following, an instance of the ReadingActivity plug is created in the eLamp (as a local reference) and a positive response is sent back to the eDesk. The instance of the ReadingActivity plug is notified for changes by its remote counterpart plug and this interaction serves as an intermediary communication channel. In case of a negative plug compatibility test, a negative response message is sent to the eDesk, while no instance of the ReadingActivity plug is created. When the eDesk receives a positive response, it also creates an instance of the Light plug, and the connection is established. Figure 5 summarizes the whole procedure.

*Synapse activation*: After connection has been established, the two plugs are capable of exchanging data. Output plugs (ReadingActivity) use specific objects, called shared objects (SO), to encapsulate the plug data to send, while input plugs (Light) use specific event-based mechanisms, called shared object listeners (SOL), to become aware of incoming plug data. When the value of the shared object of the ReadingActivity plug changes the instance of the Light plug in the eDesk is notified and a synapse activation message is sent to the eLamp. The eLamp receives

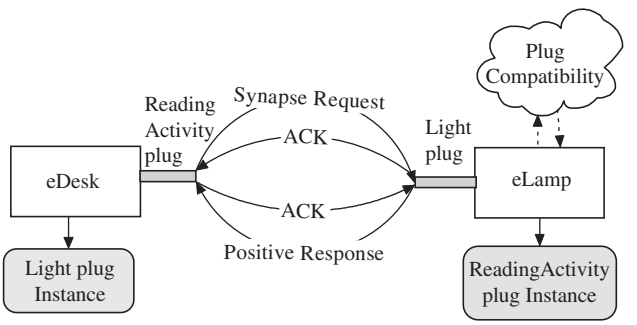


**Fig. 5.** Synapse establishment between plugs Reading and Light_Switch.

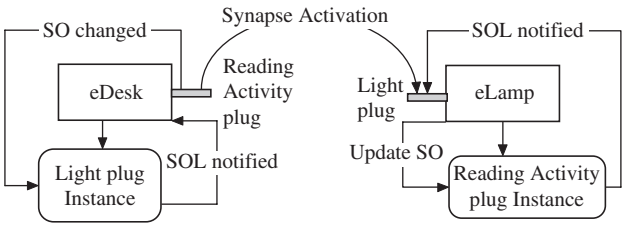


**Fig. 6.** Synapse activation.

the message and changes the shared object of its ReadingActivity plug instance. This, in turn, notifies the target Light plug, which reacts as specified (Fig. 6).

*Synapse disconnection*: Finally, if one of the two connected plugs breaks the synapse, a synapse disconnection message is sent to the remote plug in order to also terminate the other end of the synapse. Synapse disconnection can be either initiated explicitly by the user, or indirectly if one of the two artifacts becomes unavailable (e.g., goes out of range, its battery fails, etc.).

### 4.2. Inter-Artifact Communication

The Communication Module is responsible for communication between different artifacts. This module, implements application-level protocols for connectionless *ad-hoc* communication as well as mechanisms for internal diffusion of information exchanged. Peer-to-peer communication is implemented adopting the basic principles and definitions of JXTA.[12] Peers, pipes, and endpoints are combined into a layered architecture that provides different levels of abstraction throughout the communication process. Peers implement protocols for resource and service discovery, advertisement, routing as well as the queuing mechanisms to support asynchronous message exchange. In order to avoid large messages and as a consequence traffic congestion in the network, XML-based messages are used to wrap the information required for each protocol. Pipes correspond to the session and presentation layers of the ISO-OSI reference model, implementing protocols for connection establishment between two peers, supporting multicast communication for service and artifact discovery, while at the same time guaranteeing reliable delivery of messages. In cases where reliable network protocols are used in the transport layer (e.g., TCP/IP), pipes are reduced to acknowledging for application-level resource availability (e.g., sending synapse request message to an incompatible plug will return a NACK message). Endpoints are considered as the fundamental networking units and are associated to specific network resources (e.g., a TCP port). According to the transport layer chosen we can have many different endpoints (e.g., IP-based, Bluetooth, IrDA, etc.), which can also serve as a bridge for different networks. Finally, in order to discover and use services and artifacts beyond the reachability of wireless protocols (e.g., RF, bluetooth), we have designed a hybrid routing protocol

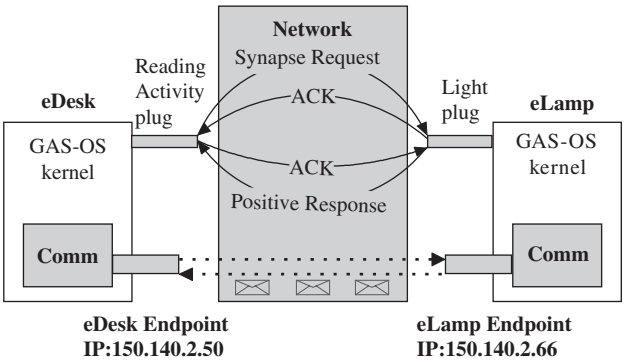


**Fig. 7.** From Plug/Synapse interactions to p2p communication.

based on ZRP.[13] Our protocol borrows the concept of zone creation and maintenance of ZRP and switches between DSR[14] and DSDV[15] inside each zone depending on certain QoS criteria, while for intra-zone routing DSR is used. Our hybrid protocol combines a proactive and a reactive part, trying to minimize the sum of their respective overheads and scales very well when the traffic or the mobility is increased.

Figure 7 shows the p2p communication between the eDesk and eLamp artifacts described in the example introduced in Section 2. Both the eDesk and the eLamp are considered to own a communication module with an IP-based (dynamically determined) Endpoint. Plug/Synapse interactions (e.g., synapse establishment) are translated to XML messages by the communication module and delivered to the remote peer at the specified IP address.

### 4.3. Interfacing with the Artifact Hardware

The interfacing of the artifact with its hardware (sensors/actuators) is performed as collaboration between the GAS-OS I/O driver and the State Variable Manager (SVM) module of GAS-OS. SVM holds two separate structures, one for the Read Only (RO) and one for the Read Write (RW) state variables. State Variables reflect the state of an artifact's hardware, like sensors and actuators. For example, the eDesk has among others, a proximity sensor to sense that a chair is near the desk, and the eLamp has one bulb actuator, both reflected inside GAS-OS as state variables in the SVM.

Through communication with the eDesk GAS-OS driver (Fig. 8) the eDesk's SVM retrieves all the sensor information of the eDesk and registers itself as a listener for changes of the environment. Moreover, it communicates with the Process Manager to promote the eDesk-eLamp communication as it feeds the ReadingActivity plug with new data coming from the hardware, which finally result in the ReadingActivity-Light synapse. On the other end of the synapse the eLamp receives data from the Light plug and through the Process Manager the data are translated to low level actuator data, resulting in the eLamp's bulb actuator.

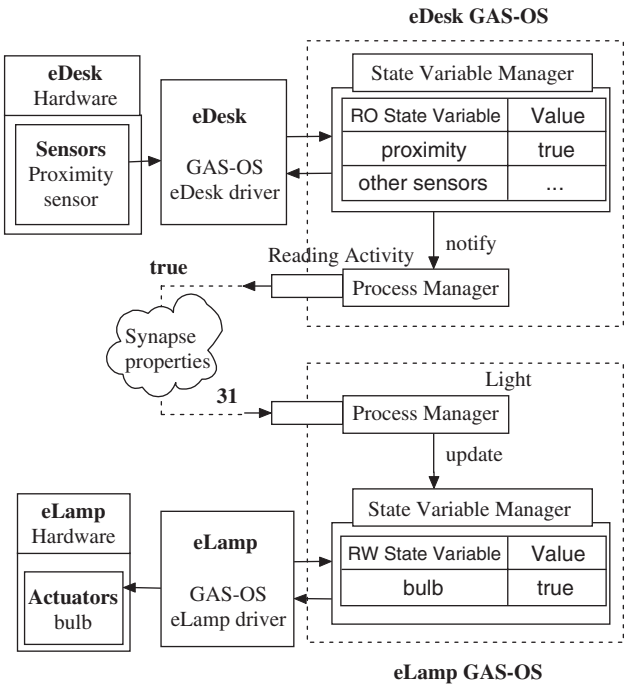


**Fig. 8.** Communication with hardware.

The matching of the true/false values of the Reading-Activity plug with the enumerated values $\{1 \ldots 31\}$ of the Light plug, is done by configuring the properties of the synapse, as we have seen in the example of Section 2. So for example by mapping the true state of the eDesk to value 31 of the eLamp and false to 0 we have the following desired behavior. The Mappings structure holds records where the key is the Synapse itself and the content is a number of values-to-states mappings. The Process Manager uses these mappings to filter the incoming information from input plugs and give a specific meaning to the incoming data.

### 4.4. Performance Evaluation

To estimate the performance of GAS-OS and its appropriateness to support the execution of ubiquitous applications, a performance and scalability analysis based on theoretical analysis as well as on experimental data was carried out. The results involve the memory requirements of GAS-OS and the throughput for the case of artifact discovery in relation to the available services (plugs). Finally a pure experimental measurement of the synapsing process and communication takes place in order to obtain an indication of the time required to set up a ubiquitous application.

The code size of the current implementation of GAS-OS is approximately 200 KB. Measuring the memory footprint is crucial in order to indicate that GAS-OS can be executed on resource constraint devices. We measured the memory footprint of the GAS-OS kernel running upon the Sun Personal Java on a Compaq IPAQ PDA reference system.

First, measurements were done by instrumented special measurement code inside GAS-OS and second using the JProbe Memory Profiler tool.[16] In both cases results seem to converge to approximately 23 Kbytes of memory, while during runtime more memory may be allocated depending on the application (e.g., number of plugs, synapses, device capabilities, etc.).

As plugs and synapses are mainly what increases memory during the execution of an application, we studied the relation between the number of plugs and the number of synapses that participate for constraint amounts of memory. Maximum memory allocation is achieved when each plug participates in only one synapse (Fig. 9(a)). The more plugs participating in one synapse, the more the allocated memory until we reach the memory constraint. From this point and on (peaks) more synapses can only be achieved if distributed to fewer plugs.

In order to measure the throughput of GAS-OS we consider the process of discovering artifacts with a certain number of plugs. Studying the discovery process gives an

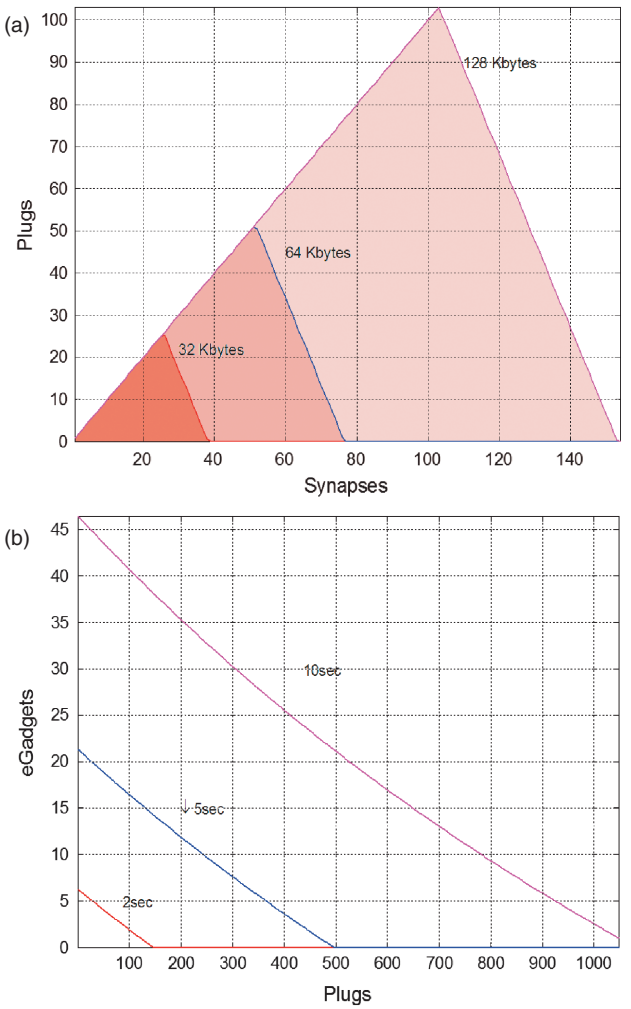


**Fig. 9.** (a) Maximum number of synapses when constraining memory versus the number of plugs that can participate. (b) Number of eGts that can be discovered in a certain period of time versus the number of plugs.

**Table II.** Synapsing and communication times.

| Time (ms) | 1st synapse | Last synapse | Data exchange |
|---|---|---|---|
| Min | 651 | 841 | 63 |
| Max | 1632 | 1422 | 396 |
| Average | 914 | 1019 | 183.2 |

Min, Max, and average times in milliseconds to create the 1st and the last synapse in a UbiComp application of 5 artifacts with a total of six synapses. After creating the 1st synapse only a few milliseconds are required to create the rest of the synapses, while the average time of approximately 1 sec for all six synapses is acceptable. For communication between 2 artifacts having a synapse, the average time is only a few milliseconds, which is acceptable for real time applications.

indication of the number of artifacts that will be discovered in a certain period of time, and as a consequence how long will the user have to wait in order to discover his ubiquitous environment. The number of artifacts that can be discovered in successive time intervals, versus the number of plugs (Fig. 9(b)): in order to have maximum performance overhead, we have to get to a large number of plugs per artifact.

Using code instrumentation, we measured the average time for making a synapse and for communicating in an application where five artifacts are inter-connected with six synapses (Table I). These measurements include the overhead of the IEEE 802.11b protocol, while messages exchanged vary from a few bytes to 1 Kbyte. Synapse times refer to the amount of time needed from the point the user specifies a synapse up to the time this synapse is completed. In cases where the artifacts, specified to form a synapse, are not aware of each other, a discovery phase is also included in the overall synapsing process. Thus, the min synapse time refers to a synapse without a discovery, while the max to a synapse with a discovery overhead. It is important that after synapses are established (UbiComp applications set-up) communication between artifacts is fast, satisfying our requirement for real time response.

## 5. RELATED WORK

Many of the existing systems focus on the use of simple input languages or metaphor-based GUI interfaces to ease the process of development for end-users who have little or no programming experience. For example, the work by Humble et al.[17] uses a "jigsaw puzzle" GUI metaphor in which individual devices and sensors are represented by puzzle piece-shaped icons that the user "snaps" together to build an application. While the familiar to our approach metaphor is comprehensible, the interactions are simplified to sequential execution of actions and reactions depending on local properties (e.g., sensor events), which limits the expressibility of the user's ideas. No emergent properties exist (i.e., taking into account other devices' properties), while the absence of rule-based logic results in very simple behaviours. Similar approaches we have observed in the work of Truong et al.[18] that provides a pseudo-natural language interface, using a fridge magnet metaphor and the browser approach of Speakeasy,[19] where components are connected using a visual editor based on file-system browsers.

Other research efforts are emphasizing on the design of ubiquitous computing architectures. In the context of the Disappearing Computer initiative, project "Smart-Its"[20] aims at developing small devices, which, when attached to objects, enable their association based on the concept of "context proximity." Objects are usually everyday devices such as cups, tables, chairs, etc., equipped with various sensors, as well as with a wireless communication module such as RF or Bluetooth. The goal is to add smartness to real-world objects in a *post-hoc* fashion by attaching small, unobtrusive computing devices to them. While a single Smart-It is able to perceive context information from its integrated sensors, a federation of *ad hoc* connected Smart-Its can gain collective awareness by sharing this information. However, the "augmentation" of physical objects is not related in any way with their "nature," thus the objects ends up to be just physical containers of the computational modules they host.

BASE[21] represents micro-kernel based middleware that is structured in multiple components that can be dynamically extended to interact with different existing middleware solutions (e.g., CORBA) and different communication technologies. While these approaches provide support for heterogeneity and a uniform abstraction of services the application programming interface requires specific programming capabilities (e.g., proxies are used as the application programming interface) to building applications. In contrast our Plug/Synapse model provides a high level programming model that even the end-user is capable of using it intuitively.

Proem[22] is a p2p platform supporting the application developer in creating and deploying applications. The objects managed by Proem are mainly electronic devices, such as PDAs and mobiles, and are abstracted as entities. Connectivity between entities is determined by proximity, while connected entities can form communities. Proem defines communication protocols that define the syntax and semantics of messages exchanged between peers, as well as an application environment including tools, APIs and runtime structures. However, Proem does not consider multi-hop mobile *ad hoc* networks, while proximity poses severe limitations in the formation of UbiComp applications.

Garlan et al. have taken a novel approach to managing ubiquitous computing environments with Project Aura.[23] Aura aims to "minimize distractions on a user's attention, creating an integrated environment that adapts to the user's context and needs." Aura's goal is to provide each user with an invisible halo of computing and information services that persists regardless of location. Each user is represented by its personal aura; when a user enters a new environment his or her aura marshals the appropriate

resources to support the user's tasks. Aura proposes a programming model for task-based computing as well as an infrastructure that moulds itself to the user's task or needs with little need for user intervention. However, the middleware is limited to provide adaptation only at the task level i.e., at the service level.

Aura is a representative work of UbiComp middleware systems trying to establish some kind of integrated, preinstalled technical infrastructure in a physical area, e.g., a room or building, often called an intelligent environment (IE), in which the user and his/her mobile devices are integrated on-the-fly when entering the area. The IE offers a huge variety of different capabilities and middleware services that can be used, once the device of the user is integrated.

Another approach of the same category is project iROS,[24] in that it considers physically bounded spaces such as offices and meeting rooms, and both of them provide low-level functionality. The iROS project aims to build a middleware for pervasive computing systems that enables platform and application portability, device extensibility, robustness and, ease of administration. The system is modeled as an ensemble of entities that interact with each other using message passing. However, iROS does not provide explicit support for application development and management; instead, they rely on service synchronization using their event heap.

Finally, traditional infrastructures like CORBA, Jini, UPNP, and the Cooltown infrastructure[25] have been used to address the needs of application developers targeting distributed systems. However, we have found that such systems are either too heavyweight to be applied to mobile hosts, or are not powerful and flexible enough to address the requirements of such systems. Furthermore, most of them are language or system dependent, and on the other hand, they try to provide as much functionality as possible, which leads to very complex and resource consuming systems, unsuitable for small devices. Finally, the focus of these infrastructures has by necessity been on the development of appropriate protocols and techniques to allow devices to discover each other and make use of the various facilities they offer. Limited consideration has been given to how inhabitants may see these devices or how they may exploit them to configure novel arrangements meeting particular household demands.

All the above have given us a glimpse of what the UbiComp-enabled future might perhaps bring. As Weiser noted in his seminal paper, we don't really know what's coming:[26] '*Neither an explication of the principles of ubiquitous computing nor a list of the technologies involved really gives a sense of what it would be like to live in a world full of invisible widgets. To extrapolate from today's rudimentary fragments of embodied virtuality resembles an attempt to predict the publication of Finnegan's Wake after just having invented writing on clay tablets. Nevertheless the effort is probably worthwhile.*'

## 6. DISCUSSION

Although our approach for composing UbiComp applications builds on the foundations of established software development approaches such as object oriented design and component frameworks, it extends these concepts by exposing them to the end-user to be used and configured in dynamic and *ad hoc* ways. In reverse to the majority of component-based models that have focused on software components with an emphasis to support the programmer our component model embraces a heterogeneous collection of artifacts in a way that is easily comprehensible by the end-users. To achieve this, composition tends to be as simple as possible, although some reduction in the expressiveness follows.

GAS-OS, the software that implements the Plug/Synapse model, can be considered as a component framework that determines the interfaces that components may have and the rules governing their composition. GAS-OS manages resources shared by artifacts, and provides the underlying mechanisms that enable communication (interaction) among artifacts. For example, the proposed concept supports the encapsulation of the internal structure of an artifact and provides the means for composition of an application, without having to access any code that implements the interface. Thus, our approach provides a clear separation between computational and compositional aspects of an application, leaving the second task to ordinary people, while the first can be undertaken by experienced 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; all people have to do is realize specific instances of the system. Composition achieves adaptability and evolution: a component-based application can be reconfigured with low cost to meet new requirements. The possibility to reuse devices for several purposes—not all accounted for during their design—opens possibilities for emergent uses of ubiquitous devices, whereby the emergence results from actual use.

In order to assess the Plug/Synapse conceptual model several user evaluations and studies have been performed including workshop demonstrations involving users[27] and a formal evaluation where users created and modified their own applications.[28] The evaluation was conducted in two phases. First an expert review was conducted in the form of a workshop. Subsequently, the cognitive dimensions framework was applied to assess how well the e-Gadgets concepts support end-users to compose and personalize their own ubiquitous computing environments.

The combined result of the evaluation indicated that the Plug/Synapse model apprehension was high among users, and most were able to utilize the system and make simple configurations by using the concepts and tools provided. The studies revealed also that technology-aware users were

able to use advanced features of the editor tool to define more complex patterns of system behavior.

The component based architectural abstraction is common in several engineering disciplines (i.e., software, buildings, etc.). Due to the properties of the digital self of eGadgets, users can conceptualize their tasks in a variety of ways, such as stimulus-desired response, rules, sequences and constraints between entities, etc. Consequently, there will always be an initial gap between their intentions and the resulting functionality of an artifact composition, which they will have to bridge based on the experience they will develop after a trial-and-error process.

Consequently, editing tools should be designed providing different levels of end user programming capabilities supporting the different levels of the technical competency of end users, and their willingness to appropriate the system. For the novice end-user the important thing is the configuration/reconfiguration of artifact collections for rapidly prototyping rather than programming UbiComp applications. In that direction, the user attempts in a trial-and-error approach rather than in formal procedural programming. Another conclusion was that when end users are able to "program" applications by adopting metaphors/building blocks, to which they can associate a meaning then it is easier to embrace the proposed high-level programming model.

The Plug/Synapse model provides a convenient abstraction for the development of small to medium sized ubiquitous computing applications. These systems are powerful enough to support everyday activities of people (such as home control, shopping entertainment, etc.,) thus we expect that most user-developed system will fall into these categories. When more complex systems must be developed (i.e., involving over a dozen interacting artifacts or more than one users), the direct management of synapses becomes difficult, as several issues now become important and demand the user's attention.

For instance, these include how can goals and tasks be distributed over artifacts, how can the distributed control be coordinated in order to insure that the overall system requirements are addressed, how can the system be configured with minimum user intervention etc. Although in principle such issues can be addressed via direct manipulation of plugs and synapses, the cognitive load imposed on the user and the extended learning curve may affect the adoption and utilization of the system. We attempt to address this problem by developing end-user tools which would provide abstractions of the applications and support semantically rich interaction. For example, an agent that could learn how users use their environment could receive user requirements and propose sets of synapses to realize the desired behaviours.

The proposed conceptual framework and software mediators (middleware, tools) have provided to our research team and others a useful medium for exploring new approaches on merging the physical and digital space in AmI environments. This happened by reusing and extending our framework to more adventure and innovative application domains examined in research projects undertaken by our group and associate colleagues. We mention our effort to create digital interfaces to nature, in particular to selected species of plants, enabling the development of synergistic and scalable mixed communities of communicating artifacts and plants by providing each plant with a GAS-compatible description of its properties and state to enable a seamless interaction in scenarios ranging from domestic plant care to precision agriculture.[29]

## 7. CONCLUSIONS

In this paper, we have presented an approach to resolve the problem of building reconfigurable ubiquitous applications out of augmented objects. Everyday objects, devices and software processes, known as artifacts, can be combined in a high level programming manner into UbiComp applications provided that they adapt to the proposed conceptual framework that is incorporated into a supporting middleware.

Towards this direction, we introduced the Plug/Synapse conceptual model inspired by the component-based application engineering paradigm that guides the development of software applications from pre-fabricated software components. The artifacts advertise their physical properties and digital services as Plugs. Furthermore, artifacts can collaborate via the establishment of communication channels between Plugs called Synapses.

A performance and scalability analysis of the GAS-OS prototype based on theoretical as well as experimental data has provided indications that our system is capable of supporting building and execution of ubiquitous applications, while a people evaluation has provided feedback in order to assess the concepts in the preliminary phases of the prototype implementation. This latter research has made several inroads in the effort to empower people to actively shape ubiquitous environments. It has demonstrated the feasibility of letting end-users architect ubiquitous environments, though significant advances are still needed in engineering enabling technology. The experiences reported in end-user evaluation sessions suggest that an architectural approach where users act as composers of predefined components is a worthy approach that can be further explored.

Ongoing work at this stage, aims at designing and developing the security plug-in to extend the functionality of GAS-OS to realizing the security policies of each artifact as well as handle privacy issues in applications. Furthermore, we are currently building a tool that provides a graphical interface for creating or changing rules, based on a node connection model. The advantage of this approach is that rules will be changed dynamically without disturbing the operation of the rest of the system and this can be done in a high-level manner. It is clear however that the

use of such a tool is addressed more to the developer or the advanced user rather to an everyday end-user.

**Acknowledgments:** The authors would like to thank all the fellow members of the DAISy group at CTI (daisy.cti.gr) for their valuable support in the specification of the Plug/Synapse model and in the development, testing, and evaluation of the underlying software system. We would like also to thank all the partners of the IST/FET eGadgets project for their input and support and the anonymous reviewers for their suggestions for improving this paper.

## References

1. Ambient Intelligence: From Vision to Reality, IST Advisory Group **(2003)**, http://www.cordis.lu/ist/istag-reports.htm
2. A. Kameas et al., An architecture that treats everyday objects as communicating tangible components. *Proceedings of the 1st IEEE Inter. Conf. on Pervasive Computing and Communications (PerCom03)*, Fort Worth, Texas **(2003)**, pp. 115–122.
3. M. Weiser, Some computer science issues in ubiquitous computing. *Communications of the ACM* **(1993)**, Vol. 36, pp. 74–84.
4. EU IST/FET e-Gadgets, project website: http://www.extrovert-gadgets.net
5. EU IST/FET Accord, project website: http://www.sics.se/accord/
6. D. A. Norman, The Invisible Computer, MIT Press **(1998)**.
7. I. Mavrommati, A. Kameas, and P. Markopoulos, An editing tool that manages the device associations. *Personal and Ubiquitous Computing J.*, ACM, Springer-Verlag London Ltd. **(2004)**, Vol. 8, pp. 255–263.
8. R. Schollmeier, A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications. *Proceedings of the 1st Inter. Conf. on Peer-to-Peer Computing* **(2001)**, p. 101.
9. A. S. Tanenbaum, M. F. Kaashoek, R. van Renesse, and H. Bal, The amoeba distributed operating system—a status report. *Computer Communications* **(1991)**, Vol. 14, pp. 324–335.
10. E. Christopoulou and A. Kameas, GAS ontology: An ontology for collaboration among ubiquitous computing devices. *International J. Human-Computer Studies (special issue on Protégé)*, Academic Press **(2005)**, Vol. 62, pp. 664–685.
11. The EJC (Embedded Java Controller) platform. EJC website: http://www.embedded-web.com/
12. Project JXTA website: http://www.jxta.org
13. M. Pearlman and Z. Haas, Determining the optimal configuration for the zone routing protocol. *IEEE J. Selected Areas in Communications* **(1999)**, Vol. 17, pp. 1395–1414.
14. D. B. Johnson, D. A. Maltz, and J. Broch, DSR: The dynamic source routing protocol for multihop wireless ad hoc networks. *Ad Hoc Networking*, edited by Charles E. Perkins, Addison-Wesley **(2001)**, Chap. 5, pp. 139–172.
15. C. Perkins and P. Bhagwat, Highly dynamic destination-sequenced distance-vector routing (DSDV) for mobile computers. *Proc. ACM SIGCOMM* **(1994)**, pp. 234–244.
16. JProbe Suite website: http://www.quest.com/jprobe/index.asp
17. J. Humble et al., Playing with the bits—user-configuration of ubiquitous domestic environments. *Proceedings of the 5th Annual Conference on Ubiquitous Computing*, Springer-Verlag, Seattle, Washington **(2003)**, pp. 256–263.
18. K. N. Truong, E. M. Huang, and G. D. Abowd, CAMP: A magnetic poetry interface for end-user programming of capture applications for the home. *Proceedings of the 6th Annual Conference on Ubiquitous Computing*, Springer-Verlag, Nottingham, UK **(2004)**, pp. 143–160.
19. W. K. Edwards, M. W. Newman, J. Sedivy, T. Smith, and S. Izadi, Challenge: Recombinant computing and the speakeasy approach. *Proceedings of the 8th Annual Inter. Conf. on Mobile Computing and Networking*, ACM Press, New York **(2002)**, pp. 279–286.
20. L. E. Holmquist et al., Smart-its friends: A technique for users to easily establish connections between smart artifacts. *Proc. UBICOMP*, Atlanta, GA **(2001)**, pp. 116–122.
21. C. Becker, G. Schiele, H. Gubbels, and K. Rothermel, "BASE—A micro-broker-based middleware for pervasive computing. *Proceedings of the 1st IEEE Inter. Conf. on Pervasive Computing and Communications (PerCom03)*, Fort Worth, Texas **(2003)**, pp. 443–451.
22. G. Kortuem and J. Schneider, An application platform for mobile ad-hoc networks. *Proc. UBICOMP*, Atlanta, GA **(2001)**, pp. 1–4.
23. D. Garlan, D. P. Siewiorek, A. Smailagic, and P. Steenkistie, Project aura: Toward distraction-free pervasive computing. *IEEE Pervasive Computing* **(2002)**, Vol. 1, pp. 22–31.
24. B. Johanson, A. Fox, and T. Winograd, Experiences with ubiquitous computing rooms. *IEEE Pervasive Computing* **(2002)**, Vol. 1, pp. 67–74.
25. T. Kindberg and J. Barton, The cooltown user experience, Technical Report HPL-2001-22, HP Labs **(2001)**.
26. M. Weiser, The Computer for the 21st Century, Scientific American **(1991)**, Vol. 265, pp. 66–73.
27. I. Mavrommati, P. Markopoulos, J. Calemis, and A. Kameas, Experiencing extrovert gadgets. *Proc. BCS HCI*, edited by H. Johnson, P. Gray, and E. O'Neil, Research Press International **(2003)**, Vol. 2, pp. 179–182.
28. I. Mavrommati, A. Kameas, and P. Markopoulos, Visibility and accessibility of a component-based approach for ubiquitous computing applications. *Proc. HCI International*, edited by C. Stephanidis and J. Jacko, Human Computer Interaction, Theory and Practice, Lawrenece Erlbaum and Associates **(2003)**, Vol. 3, pp. 178–182.
29. C. Goumopoulos, E. Christopoulou, N. Drossos, and A. Kameas, The plants system: Enabling mixed societies of communicating plants and artefacts. *Proceedings of the 2nd European Symposium on Ambient Intelligence*, Springer-Verlag, Eindhoven, The Netherlands **(2004)**, pp. 184–195.